Computer Science — University of Amsterdam

Bachelor Computer Science

UvA University of Amsterdam

Engineering a Domotics Application with PSF

Daan Staudt

June 9, 2008

Supervisor(s): Bob Diertens (UvA) & Inge Bethke (UvA)

Signed:

Abstract

Many paradigms exists to offer software engineers guidance during some stage of an otherwise ill-defined software engineering process. We discuss the usefulness of the software engineering process with PSF which consists of clearly defined stages, from architecture specification to implementation, each describing an application in a lower level of abstraction then the previous stage. We examine each stage of the process in detail by engineering a Domotics application which we will also extend. We conclude that the software engineering process with PSF is useful, but do offer some suggestions for improvement.

Contents

CHAPTER 1

Introduction

In the modern world of software engineering many tools and paradigms have been developed to aid the software engineer. Although tools and paradigms like OOP, UML and Rapid Prototyping offer engineers guidance during the implementation of software, few tools exist that allow an engineer to validate his implementation. Software engineering is much more then just implementing a piece of software, it starts with determining requirements, followed by writing a specification. In this report we discuss PSF [MV90, MV93], an algebraic means of writing such a specification. By writing the specification in a formal manner, rather then using natural language, it now becomes possible not only to validate the specification, but also to simulate and even animate the specification [Die00, Die97].

Many problems faced by software engineers are related to the ever-changing desires of the customer, even if a piece of software is developed in an academic setting. As soon as the customer changes the requirements the specification needs to be examined to determine the components that are affected by this change. Because the specification in traditional software engineering is written in a natural language and is monolithic in nature, finding all these components can be tricky. With the software engineering process using PSF [Die05, Die06] updating the specification after an added or modified requirement becomes easier.

In this thesis we will attempt to determine the usefulness of the software engineering process with PSF by engineering a Domotics application. Domotics, or home automation, is a system for centralized control of the various aspects of a house, from garage doors to lights to home entertainment. Domotics applications have proven useful examples in the past for illustrating tools and paradigms that exploit or facilitate concurrency.

1.1 Process Specification Formalism

PSF is an algebraic specification formalism for specifying concurrent processes based on ACP, the Algebra of Communicating Processes [BK87]. For specifying the data types these processes operate on PSF contains elements borrowed from ASF, the Algebraic Specification Formalism [BHK89]. For our Domotics application we will not make much use of the data type description capabilities of PSF, instead we focus on the process description capabilities. The PSF operators that describe processes are the same as the ACP operators, but PSF has been extended over the years to include for example interrupts [Die94] and iteration [DP94]. PSF has all the same axioms and action relations as ACP.

PSF process specification files are split up into a number of sections. For our application we are mainly interested in the `imports`, `atoms` and `definitions` sections. In the `imports` section all the process and data specifications referenced by the specification are listed. The `atoms` section contains a list of all the atoms of the specification. The process definitions are listed in the `definitions` section. Process definitions in PSF may be recursive, even infinitely so.

1.2 The PSF Toolkit

To facilitate the use of PSF with a software engineering process the PSF Toolkit [Vel93, Vel95] has been created. This Toolkit contains programs to simulate and animate PSF specifications.

To simulate PSF specifications the Toolkit provides the `sim` program. After a specification has been compiled into a machine readable format it can be passed as a parameter to this program. When `sim` is started we must first select the process definition we want to simulate. Then for each discrete time step we can choose which atom is to be executed. We can also let the simulator randomly execute possible atoms.

For animating PSF specifications the `anim` program is provided. It examines the specifications to generate a visual representation of the defined processes as a graph. The animator will mostly be used in conjunction with the simulator. The Toolkit provides a wrapper to execute both, called `simanim`. In that case the simulator will provide the animator with a list of possible atoms and the animator will let the user choose. Colors are used to indicate to the user which processes can execute an atom. The atoms themselves are listed in a pop-up list, one for each process.

In some cases it is desirable to record the atoms executed, for example to show a series of actions leading to deadlock. The Toolkits simulator allows users to trace a specifications execution and store this trace to a file. The trace file can then be passed as an argument to the animator to be animated.

1.3 The ToolBus

Developed at the CWI (Centrum voor Wiskunde en Informatica) in Amsterdam the ToolBus [BK98] shown in Figure 1.1 is a software coordination architecture. It coordinates all communication between the various components, or tools, that make up an application. All communication between these tools must take place through the ToolBus. If tools were allowed to communicate outside of the ToolBus, that would undo the advantages of having a coordination architecture in the first place.

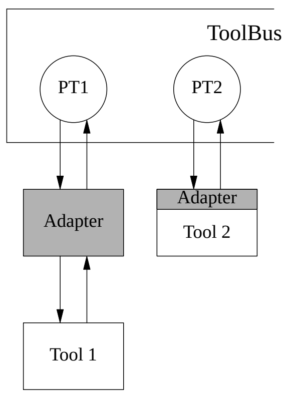


Figure 1.1: A diagram showing the ToolBus. The PT1 and PT2 processes exist inside the ToolBus and communicate with Tool1 and Tool2 outside the ToolBus. The tools do not communicate outside the ToolBus. (Figure taken with permission from B. Diertens.)

A ToolBus application consists of two types of processes, ToolBus processes and tools. ToolBus processes exist inside the ToolBus and are formally defined in a 'T script', see Section 1.3.1. Tools exist outside of the ToolBus and can be written in a variety of languages, such as Tcl/Tk [Ous94]. Tools can either be connected directly to the ToolBus if a language wrapper is available, or indirectly if a user has written his own language adapter. The communication protocol for communication between ToolBus processes and tools and for ToolBus processes among themselves is described in Section 1.3.2.

1.3.1 T script

The processes inside the ToolBus that make up the coordination architecture are described in a so-called 'T script'. A T script is made up of several process descriptions and commands for starting tools. The syntax and grammar of the process descriptions resemble that of PSF. There are some notable differences, however. For example where PSF supports infinite recursion, T script does not. Because infinite recursion is merely some syntactic sugar to split up long process definitions into several smaller ones, any PSF specification can be rewritten to use the BKS (Binary Kleene Star) operator and be copied almost verbatim into a T script.

1.3.2 The ToolBus communication protocol

All communication between tools and the ToolBus uses the atoms listed in Table 1.1. These atoms take arguments of the `TBterm` and `TBid` types. A `TBterm` can be any term used by processes inside the ToolBus, for example data or component identifiers. A `TBid` is a unique identifier of a tool outside the ToolBus.

Tool action	ToolBus process action
`TBsend "snd-value(TBterm)"`	`rec-value(TBid, TBterm)`
`proc foobar(TBterm)`	`snd-eval(TBid, foobar(TBterm))`
`proc foobar(TBterm)`	`snd-do(TBid, foobar(TBterm))`
`TBsend "snd-event(TBterm)"`	`rec-event(TBid, TBterm)`
`TBsend "snd-event(TBterm, TBterm)"`	`rec-event(TBid, TBterm, TBterm)`
`proc rec-ack-event(TBterm)`	`snd-ack-event(TBid, TBterm)`

Table 1.1: The atoms used by the ToolBus processes and tools written in Tcl/Tk for communication. The `proc` keyword indicates a procedure. A ToolBus process sending information to a tool will result in a procedure call in the tool. Names of the communication atoms have been omitted as they are never seen by the engineer.

The ToolBus enforces a strict communication protocol when using these atoms. Tools are only allowed to communicate unsolicited information to a ToolBus process if they send it as an event, using the `snd-event()` atom. Every time a tool sends an event to a ToolBus process, that process must at some point in time, but before the tools sends another event, acknowledge the event using the `snd-ack-event()` atom. Failure to do so will result in a protocol error.

ToolBus processes can send information to a tool using either the `snd-do()` or the `snd-eval()` atom. A tool may not respond directly to a ToolBus process that executed a `snd-do()`, of course it may always do a `snd-event()`. A tool must always respond to a ToolBus process that executed a `snd-eval()` by executing a `snd-value()`. Failure to do so again results in a ToolBus protocol error.

ToolBus processes communicate among themselves using the `snd-msg()` and `rec-msg()` atoms. These atoms take three arguments, a `TBterm` identifying the sender, a `TBterm` identifying the receiver and a `TBterm` containing the message.

1.4 Software engineering with PSF

The software engineering process breaks down into several steps, when we use PSF we can identify five distinct steps. Firstly we must determine, or are given, a list of requirements (Section 1.4.1). Secondly we turn these requirements into a list of scenarios (Section 1.4.2). Thirdly we specify an architecture for the application (Section 1.4.3). Then we turn this architecture specification into an application specification (Section 1.4.4). Finally we use the application specification as a template to create an implementation (Section 1.4.5). During the architecture and application specification phases we make use of the software engineering pack for PSF described in Section 1.4.7 and validate our specifications using the technique described in Section 1.4.8.

1.4.1 Requirements

The first step in any engineering process is to determine what the product to be developed actually needs to do. These requirements are often given to the engineer together with background information about the customer, so that the engineer understands the context within which the requirements are set. The requirements are usually given in natural language, although they may include references to more formal specifications. One of the requirements for a web server application might for example be that it supports the TCP/IP protocol, in which case a reference is made to a formal specification of that protocol, an RFC.

1.4.2 Scenarios

After the requirements for an application have been determined we use them to create so-called scenarios for the application. A scenario is a description of one specific function of an application, without references to any other function. A scenario for a web server application might be that it serves a user with a page. Note that there are no references to other functions, such as having a user connect.

1.4.3 Architecture specification

Now that we have a list of scenarios to guide us through the rest of the engineering process, we can begin specifying our application. This is the first time in the engineering process we encounter PSF. The grammar and syntax of this algebraic specification language for describing the behavior of concurrent processes has been discussed in Section 1.1.

An architecture consists of one or more components that communicate data. The PSF software engineering pack, see Section 1.4.7, contains an architecture library that formally describes components and data in an abstract sense. Components are processes that have been given an identifier. Data can be any defined data type, although when doing software engineering with PSF it is best to keep data as abstract as possible, i.e. not to specify data types explicitly.

For communicating data the processes that are the components of the architecture may use the library-provided `snd()` and `rec()` atoms, with `comm()` as the communication atom. The arguments for these atoms are the identifier of the source component, the identifier of the destination component and the data, in that order. For example to send a message from a sender to a receiver, the sender would execute `snd(sender >> receiver, message)` and the receiver would have to execute `rec(sender >> receiver, message)` the communication would be `comm(sender, receiver, message)`.

1.4.4 Application specification

The next step in our engineering process is to turn our architecture specification into an application specification. Where the architecture specification deals with the components of a system, the application specification splits these components into a coordination architecture and the tools that connect to it. The coordination architecture is made up of processes that communicate as was previously defined in the architecture specification.

Because we will implement the application as a ToolBus application our application specification will use an abstract version of the ToolBus communication protocol defined in Section 1.3.2. The application library from the PSF software engineering pack provides the communication atoms listed in Table 1.2. The arguments, `TBterm` and `TBid`, have the same meaning as those described in Section 1.3.2.

The ToolBus atoms `snd-msg()` and `rec-msg()` used for communication between processes inside the ToolBus are modeled by the `tb-snd-msg()` and `tb-rec-msg()` atoms, taking the same arguments.

Tool action	ToolBus process action
`tooltb-snd(TBterm)`	`tb-rec-value(TBid, TBterm)`
`tooltb-rec(TBterm)`	`tb-snd-eval(TBid, TBterm)`
`tooltb-rec(TBterm)`	`tb-snd-do(TBid, TBterm)`
`tooltb-snd-event(TBterm)`	`tb-rec-event(TBid, TBterm)`
`tooltb-snd-event(TBterm, TBterm)`	`tb-rec-event(TBid, TBterm, TBterm)`
`tooltb-rec-ack-event(TBterm)`	`tb-snd-ack-event(TBid, TBterm)`

Table 1.2: The atoms representing those that will be used by the ToolBus processes and tools for communication. The difference between `tb-snd-do` and `tb-snd-eval` is that a ToolBus process expects a reply to a `tb-snd-eval`, but not to a `tb-snd-do`. Names of the communication atoms have been omitted for the sake of brevity.

1.4.5 Implementation

Now that we have an application specification, we can make an implementation. The coordinating processes described in Section 1.4.4 can easily be transformed into a T script for the ToolBus, as described in Section 1.3.1. The tools we specified must be implemented as ToolBus tools.

We restrict ourselves to discussing ToolBus tools written in the Tcl/Tk language. In order for our Tcl/Tk tools to receive messages from the ToolBus they must have procedures that correspond to the message to be received. For example if a ToolBus process sends a message `foobar(t)`, where `t` is a ToolBus term, it results in a procedure call within the tool. The procedure that will be called is `foobar` and `t` will be the argument supplied to the procedure. The ToolBus will generate a warning if a tool is started that does not have the procedure signatures to handle all possible incoming communication.

In order for a tool to send a message to its corresponding ToolBus process it calls the `TBsend` procedure, provided by the ToolBus language wrapper for Tcl/Tk with the appropriate communication atom as its argument.

It is possible to use rapid prototyping techniques when implementing ToolBus tools because any received messages can be easily ignored and possible responses can be hard-coded.

1.4.6 Feedback

An important part of any engineering process is feedback. As an application is described at decreasing levels of abstraction, flaws or deficiencies at higher abstraction level descriptions are identified. In software engineering this means that if a flaw or corner-case is discovered during implementation the specification should be adapted.

When using PSF for software engineering engineers can validate their specifications, see Section 1.4.8. As a result it may be more likely that the specification will be updated then it would be when not using PSF to do software engineering. Updating the specification rather then trying to circumvent the corner-case in the implementation may make it easier to maintain or extend an application once it has been built.

This feedback does not exist just between the specification and the implementation, it exists between all the steps of the software engineering process. For example difficulties encountered during the application specification phase may lead to changes in the architecture specification.

1.4.7 The PSF software engineering pack

The software engineering pack for PSF consists of several libraries to be used during the architecture specification and application specification phases of the engineering process. These libraries contain specifications of atoms to be used for communication between processes, described in Sections 1.4.3 and 1.4.4, as well as environments for executing architecture and application specifications. In practice these environments merely provide the specification the means to signal it has completed its execution and can be shut down.

The PSF software engineering pack also contains several tools that can be used to automatically convert an architecture specification into an application specification. This trajectory makes use of both vertical and horizontal implementation techniques to convert an application specification to an architecture specification. For a detailed description of these techniques see [Die07].

Figure 1.2 shows the trajectory of transforming an architecture specification into an application specification. First the `se-refine` program is applied, which maps the abstract atoms of the architecture specification to concrete atoms for the application specification. This is called vertical implementation and requires the engineer to specify the mappings in a mappings file.

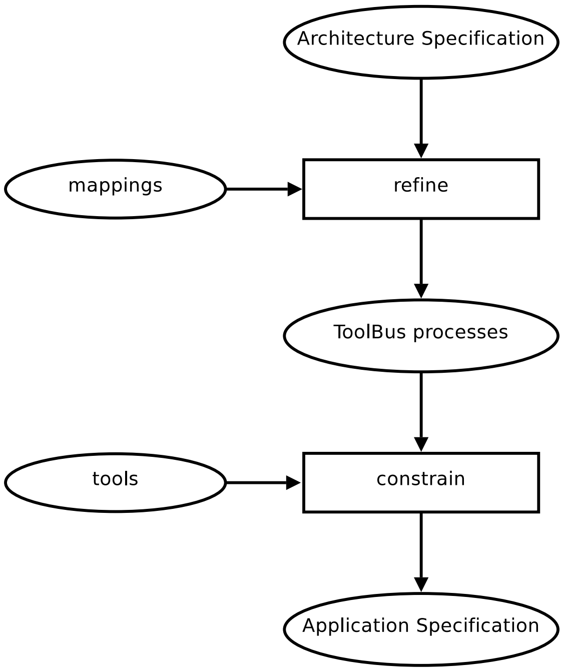


Figure 1.2: A diagram showing the trajectory of converting an architecture specification into an application specification using the programs provided by the software engineer pack for PSF.

We then run the `se-constrain` program to constrain the processes representing ToolBus processes with the processes representing tools we specified in a tools file. This is called horizontal implementation and the result is a constrained ToolBus application specification

1.4.8 Validation

Using the `simanim` program described in Section 1.2 we can simulate and animate our architecture specifications and our application specifications. This allows us to visually validate our specifications.

During the architecture specification we can validate our specification against the scenarios we have written. Each time we expand our architecture specification to support yet another scenario we use the `simanim` program to step through our specification. We make sure that all the previous scenarios are still supported, i.e. there is no functional regression. Because we are designing systems with concurrent processes we try to think of situations which might cause deadlock and simulate those. If we have designed our system well no deadlock should occur. We also use the random function of the simulator to randomly execute possible atoms to see if deadlock occurs. Although this does not formally prove that deadlock cannot occur, with small systems it does provide a general indication of the stability of the application.

When we have converted our architecture specification to an application specification we validate our application specification against the architecture specification. Because we have constrained our ToolBus processes with tools we must ensure the application specification still supports all the scenarios the architecture specification did. Now that we have added more components to our system we have introduced more possibilities for deadlock to occur. We again examine several possible situations where deadlock might occur.

CHAPTER 2

Engineering the Domotics application

We will demonstrate the usefulness of PSF for engineering software by engineering a Domotics application. For this application, as with any application, we follow the five steps of the PSF software engineering process as described in Section 1.4.

2.1 Requirements

Because we are not engaged in software engineering in a commercial environment, we have the privilege of writing our own requirements. Since we plan to extend our Domotics application in the future, to demonstrate how the engineering process with PSF stands up to changing requirements, we have kept the initial requirements relatively simple.

1. The system must provide an interface for driving actuators.
2. The system must provide an interface for the user to control the settings of all the automated equipment.
3. The system must provide an interface for the presentation of actuator data to provide the user with an overview of the house.
4. The system must be able to simulate components connected to the interfaces described in 1 - 3 to operate without hardware.

The list above enumerates our four requirements. They have deliberately been kept as abstract as possible to demonstrate in the following chapters how the software engineering process with PSF allows engineers to experiment with specifications to find an optimal specification in a vast space of possible specifications.

2.2 Scenarios

From the requirements listed above we can deduce that users set actuators contained within the house. Actuators are not specifically defined, merely mentioned as ‘automated equipment’. The system must also provide some sort of display so that the user may see the state of the house. The six scenarios listed below are what we came up with.

1. Set single actuator
   The user sets an actuator to a desired state using a control interface. The result is also shown on a display.
2. Hardware triggered actuators
   The state of an actuator can also be changed by using a physical medium, e.g. a light switch.

3. Multiple users
   More then one user can set actuators, either using the control interface or a physical medium.

4. System reset
   A user resets the system. All the actuators are set to their default states.

5. Create and delete preset
   A user records a series of actuator state changes to use as a preset to be applied later.

6. Apply preset
   A user changes the state of a series of actuators with a single command using the control interface.

The scenarios are fully independent, although it may not be immediately clear. While ones instincts might say, for example, that a preset cannot be applied if it has not yet been created, the architecture required to apply a preset may very well have no dependencies on the architecture required to create a preset. As such the two scenarios are independent. In fact it is desirable to keep the dependencies in such a case to a bare minimum, as this will make it easier to modify the architecture for a specific scenario without requiring changes to those for the other scenarios.

2.3 Architecture Specification

Now that we know which tasks the application must perform we must design an architecture for it. An architecture consists of several components, specified as processes that communicate. We begin by specifying an architecture capable of performing the task described in the first scenario and then extend it to perform the next and so on.

2.3.1 Setting an actuator

We start our architecture specification by looking at the first scenario, a user setting an actuator. It is clear that the architecture for the first scenario will require at least three components (see Section 1.4.3): a user, a display and the house.

The data that we need for this architecture must contain enough information to determine which actuator is to be set and to which setting. We call this datum `actuator-state` and we do not specify what it contains. The reason is that we do not wish to limit the possible implementations to only a house containing a predefined number of actuators each with a number of predefined states.

In the case of specifying a system that does require the data to be specified in great detail, e.g. a physics simulator requiring the gravitational constant, PSF provides a vast library [vW93] of specifications for data types such as booleans, naturals and integers.

Having determined the required components and data types, we now specify their behavior. The user first makes it known he wants to set an actuator, after which the user component of the architecture communicates the desired state to both the house and the display. The display receives an actuator state from the user and updates a picture of the house to show the appropriate actuator in the appropriate state. The house receives an actuator state from the user and sets the actuator to the desired state.

Figure 2.1 shows the animation of the architecture specification for this scenario. The animation was generated by the `simanim` program from the PSF Toolkit, see Section 1.2. The oval shapes represent architecture components. The inner box surrounds the components of the Domotics architecture, the outer box surrounds the environment provided by the architecture library of the software engineering pack for PSF, see Section 1.4.7. In practice the ArchitectureControl and ArchitectureShutdown components only serve to simulate the end of the applications execution. The components represented by the darker ovals can execute an atom at this point in time. The arrows going from one oval to another indicate that the components can communicate, the direction of the arrow indicates the direction of the communication.

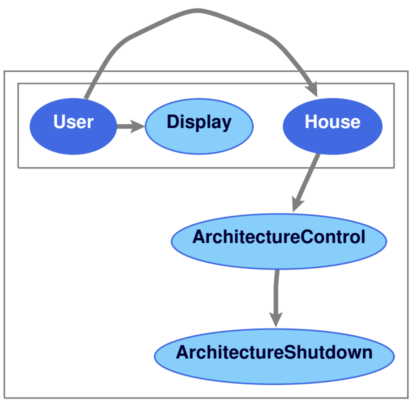


Figure 2.1: An animation of the architecture specification for the first scenario of the Domotics application.

The listing below shows the process definition of the User process, which takes the role of the `user` component in our architecture. Please note the `() * delta` construction which uses the BKS (Binary Kleene Star) operator together with deadlock to ensure that the subprocess between parentheses is infinitely repeated. The animation in Figure 2.1 shows the state of the architecture right after it has been started, before any process has executed any atoms. The oval representing the `user` component is dark because the User process can execute the `push-set-actuator` atom at this point in time. There are arrows going from the `user` component to the `house` and `display` components because of the communication atoms on lines 4 and 5 in the User process definition.

```
1  User =
2    (
3      push-set-actuator .
4      snd(user >> house, actuator-state) .
5      snd(user >> display, actuator-state)
6    ) * delta
```

2.3.2 Hardware triggered actuators

We now move on to the second scenario, hardware triggered actuators. Rather then specifying all components again from scratch, we modify the components we specified for the first scenario to accommodate the second. There is no need to add additional components or data types to accomplish this. All we need to do is specify more communication behavior. In particular we need the house to send an `actuator-state` to the user, and the user to forward that to the display.

The reason we choose not to let the house communicate directly with the display is that we want to keep the interface, or communication channels, of the house as simple as possible. That way we are not forced to make more and more assumptions about or impose more and more constraints on the physical house, i.e. the implementation of the house.

Figure 2.2 shows the updated version of the architecture for our Domotics application which now supports both the first and the second scenario. The number of components has remained the same, however we have now added more communication. There is now an arrow going from the `house` component to the `user` component, indicating that the house can at some point in time communicate information to the user.

The listing below shows that we have made some changes to the processes themselves. Comparing this definition of the User process with the one from the first architecture we see that main difference is the inclusion of the communication atoms on lines 3 and 8. We have also had to use the BKS operator on line 11 to prevent deadlock in the case the house is trying to send an updated actuator state at the same time the user is.

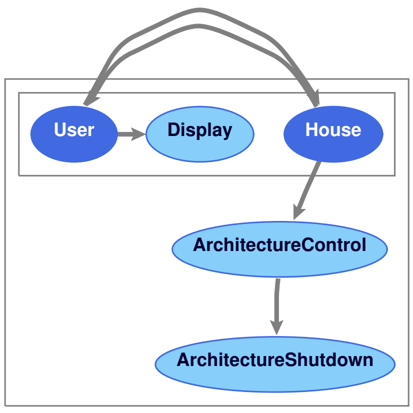


Figure 2.2: An animation of the architecture specification for the second scenario of the Domotics application.

```
User =
  (
    rec(house >> user, actuator-state) .
    snd(user >> display, actuator-state)
  + push-set-actuator .
    (
      (
        rec(house >> user, actuator-state) .
        snd(user >> display, actuator-state)
      )
       *
      (
        snd(user >> house, actuator-state) .
        snd(user >> display, actuator-state)
      )
    )
  ) * delta
```

2.3.3 Multiple users

The third scenario calls for there to be multiple users. Because we want the interface to the house to remain as simple as possible we suggest the following. Imagine our Domotics architecture as a tree with the house as the root. The house only communicates with one other component, called a user manager. That user manager can in turn communicate with two other components, either users or more user managers. This tree can be expanded to have any number of users $n \geq 2$. Because each user must have his own display we say that the first user manager, the one connected to the house, also communicates with a display manager component at the root of a tree with display managers and displays.

Having to specify a separate process for each component as we have done so far would become very time consuming with large numbers of users. Therefore we use a technique called parametrization. Instead of explicitly incorporating the component names into the process definitions, we make the component identifiers parameters of the definitions.

Let us demonstrate parametrization by looking at the process definition of a user. The listing below shows the parametrized process definition of the User process. In this process definition the terms `self` and `manager` are parameters for component identifiers. Note that we have also parametrized the atom `push-set-actuator` (more about that in Section 4.1). Note that user components no longer need to deal with hardware triggered actuators, the first user manager component forwards this information to the first display manager component.

```
User =
  (
    push-set-actuator(self) .
    snd(self >> manager, actuator-state)
  ) * delta
```

To create a component called `user1` which is connected to a user manager component called `usermanager` we need to bind the parameters of the User process to the component identifiers we have defined for our architecture. The listing below shows how this is done.

```
User {
  User-Parameter bound by [
    self -> user1,
    manager -> usermanager
  ] to DomoticsData
  renamed by [
    User -> User1
  ]
}
```

We have given the parameters of the User process a name, User-Parameter, and we use the PSF construct `bound by` to bind them to the component identifiers we have defined in the DomoticsData data module. We also need to use the PSF construct `renamed by` to rename the User process so that we can have multiple instances running in parallel. Figure 2.3 shows our parametrized architecture with two users.

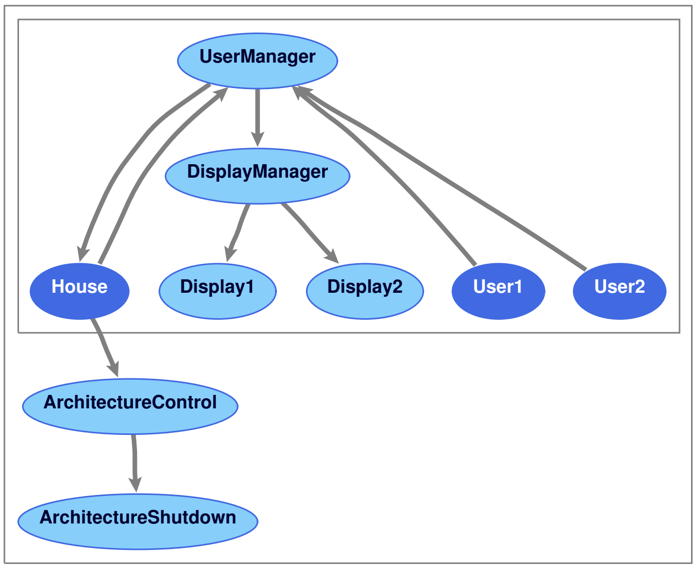


Figure 2.3: An animation of the architecture specification for the third scenario of the Domotics application. The animation of the architecture for the fourth scenario is identical to this one.

2.3.4 System reset

Incorporating the fourth scenario, a system reset, into our architecture does not require much work. All we need is a new datum to represent a request for a reset, `reset-please`. The modifications to our existing components are minimal as can be seen in the listing below. We simply specify how and when the components communicate `reset-please`. Naturally this must introduce some new atoms in the user, display and house components.

The animation of the architecture for this scenario looks exactly the same as the animation of the architecture for the third scenario, see Figure 2.3.

The listing below shows the parametrized definition of the DisplayManager process. In the architecture with two users, and thus two displays, shown in Figure 2.3 the `self` parameter is bound to `displaymanager`, the `manager` parameter to `usermanager`, the `child1` parameter to `display1` and the `child2` parameter to `display2`.

```
DisplayManager =
  (
    rec(manager >> self, actuator-state) .
    snd(self >> child1, actuator-state) .
    snd(self >> child2, actuator-state)
  + rec(manager >> self, reset-please) .
    snd(self >> child1, reset-please) .
    snd(self >> child2, reset-please)
  ) * delta
```

2.3.5 Creating and deleting presets

Expanding our architecture specification to accommodate the fifth scenario, creating and deleting presets, is somewhat more cumbersome. First we explain the semantics we will use for creating presets. We feel it is useful for users to be able to create a preset containing the state of an arbitrary number of actuators, as opposed to all of the actuators. We leave it up to the implementation to decide if a preset containing no actuator states is still a preset.

To create a preset a user can press some 'record preset' button on his remote control, then set any number of actuators to a certain state and then press some 'store preset' button to store the preset to be applied later. Naturally we must also provide functionality to discard a preset after the user has begun recording it. It would seem useful for the user to see the effect that a preset he is recording will have, so we also set each actuator to the desired state while the user is recording a preset. Figure 2.4 shows the updated architecture to include the fifth scenario.

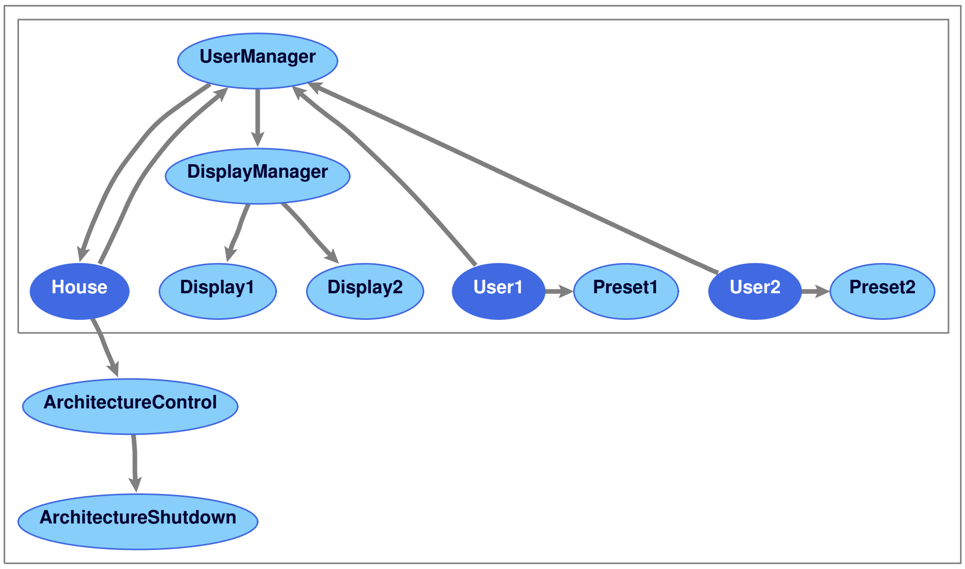


Figure 2.4: An animation of the architecture specification for the fifth scenario of the Domotics application.

We will need to introduce several new types of data to facilitate these semantics, namely record-preset, store-preset, discard-preset and delete-preset. To minimize the impact on existing communication channels we have decided that the preset components only communicate with their respective user component.

The listing below shows the parametrized process definition of the Preset process. The semantics of our presets are evident from the process definition. Once recording has begun, the BKS operator on line 10 enforces that actuator states are received until either the store-preset or the discard-preset datum is received.

```
 1  Preset =
 2    (
 3      rec(user >> self, record-preset) .
 4      start-recording(self) .
 5      (
 6        (
 7          rec(user >> self, actuator-state) .
 8          store-actuator(self)
 9        )
10         *
11        (
12          rec(user >> self, store-preset) .
13          store-preset(self)
14        + rec(user >> self, discard-preset) .
15          discard-preset(self)
16        )
17      )
18    + rec(user >> self, delete-preset) .
19      delete-preset(self)
20    ) * delta
```

2.3.6 Applying presets

Since a preset is nothing more then a list of actuators and their desired states, the application of a preset consists of a series of actuator-state data to be communicated to the house. Once the user has sent an apply-actuator datum to the preset it responds by sending zero or more actuator-state data, followed by the preset-applied datum. Figure 2.5 shows the animation of our final Domotics architecture.

This means that if an implementation of a user would try to apply a non-existing preset the only sensible thing an implementation of a preset could do is to immediately reply with preset-applied. This makes a fine example of how software engineering with PSF helps implementers to deal with corner-cases. The listing below shows part of the process definition of the Preset process, in particular the part that deals with applying presets.

```
Preset =
  (
    [...]
  + rec(user >> self, apply-preset) .
    start-applying(self) .
    (
      (
        determine-next-actuator(self) .
        snd(self >> user, actuator-state)
      )
       *
      (
        preset-applied(self) .
        snd(self >> user, preset-applied)
      )
    )
  + [...]
  ) * delta
```

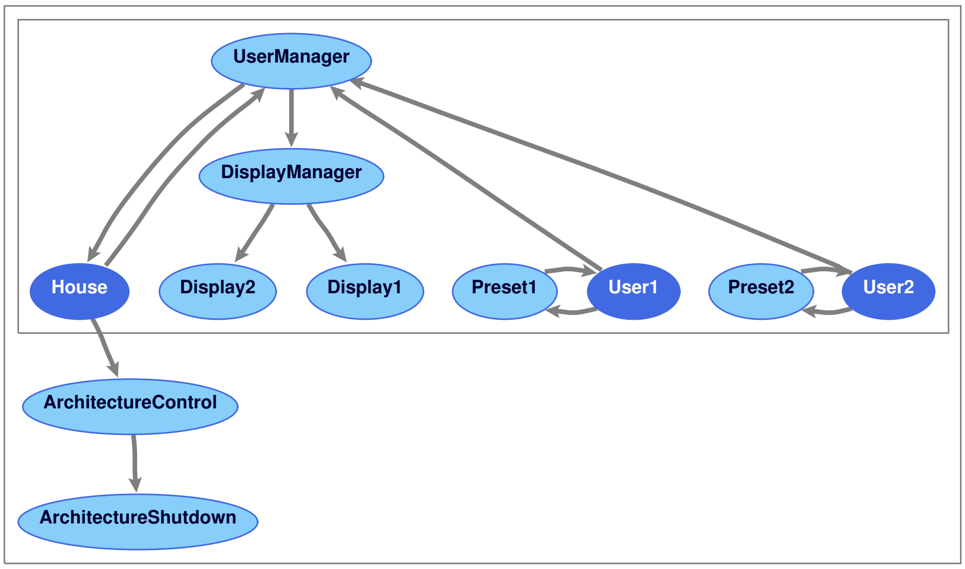


Figure 2.5: An animation of the architecture specification for the sixth scenario of the Domotics application.

2.4 Application Specification

In the application specification stage we split all our architecture components into two parts. One part will become part of the coordination architecture, while the other will be replaced by a real tool during implementation. As mentioned in Section 1.4.7 the software engineering pack for PSF contains several programs to aid in transforming an architecture specification into an application specification. These rudimentary ad-hoc tools were created to relieve the burden of manually rewriting the existing architecture specification.

We start by taking our architecture components and running the `se-refine` program. This program refines atoms in the components based on mappings defined in a file by the engineer. For example the `push-set-actuator` atom of the `user` component will be refined to a `tb-rec-event()` followed by a `tb-snd-ack-event()`. This means that the user component instead of deciding by itself an actuator needs to be set will receive an event from its tool indicating this needs to be done. A mappings file will also contain mappings from the `snd()` and `rec()` atoms used for communication between architecture components to the `tb-snd-msg()` and `tb-rec-msg()` atoms from Section 1.4.4 respectively.

The listing below shows part of the mappings file. The first two lines tell the `se-refine` program to turn all the `snd()` atoms into `tb-snd-msg()` atoms and all the `rec()` atoms into `tb-rec-msg()` atoms. Lines 5 through 8 refine the `push-set-actuator` atom. Because our application specification is parametrized, this atom takes a parameter `self` indicating the component identifier. Because each component in the coordination architecture must communicate with its own tool we introduce the `SELF` parameter to indicate the identifier of the tool. Because tool identifiers are of the `TBid` type, we write the parameter in all caps to more easily differentiate between them and parameters of the `TBterm` type with the same name.

```
1 default snd($1 >> $2, $3) -> tb-snd-msg($1, $2, tbterm($3))
2 default rec($1 >> $2, $3) -> tb-rec-msg($1, $2, tbterm($3))
```

```

[...]
push-set-actuator(self) -> (
  tb-rec-event(SELF, tbterm(actuator-state)) .
  tb-snd-ack-event(SELF, tbterm(actuator-state))
  )
[...]
```

After we have refined our atoms we apply some more modifications to the components using the `se-modify` program. This program requires a file containing the modifications written by the engineer. For example we prefix all the names of our processes with the letter ‘P’ so we can easily distinct between ToolBus processes and tools, which we will prefix with the letter ‘T’.

We now write the process specifications of our tools. Because tools only communicate with their respective ToolBus processes these specifications are often much simpler then those of the ToolBus processes.

Having written the specifications for the tools, we now constrain them with their ToolBus process counterparts. This is done automatically using the `se-constrain` program, which will prefix the names of the constrained processes with the letters ‘PT’. Because we have parametrized our application to allow multiple user, see Section 2.3.3, we must skip this step and manually write the PSF specification describing the constrained processes.

The listing below shows the process definition for a constrained Display process. This particular Display process will fulfill the role of the `display1` component in the coordination architecture and will be communicating with its tool, `DISPLAY1`. The actual constraining takes place on line 21 where we state the PT-Display1 process is the PDisplay1 process in parallel composition with the T-Display1 process.

```
 1  process module PT-Display1
 2  begin
 3      [...]
 4      imports
 5        PDisplay {
 6          Display-Parameter bound by [
 7            self -> display1,
 8            SELF -> DISPLAY1,
 9            parent -> displaymanager
10          ] to DomoticsData
11          renamed by [
12            PDisplay -> PDisplay1
13          ]
14        },
15        TDisplay {
16          renamed by [
17            TDisplay -> TDisplay1
18          ]
19        }
20      definitions
21        PT-Display1 = PDisplay1 || TDisplay1
22
23  end PT-Display1
```

Now that we have specified all our processes we use the `se-gen-tb` program to generate a specification of the application where all these processes run in parallel. Figure 2.6 shows an animation of the final application specification. Note that the letters we prefixed to the process names make it easier to distinct between ToolBus processes and tools. Also note that the constrained ‘PT’ processes, the boxes drawn around each pair of ‘P’ and ‘T’ processes, map to our original architecture components.

An important observation to be made is that not all ToolBus processes will have tools. Some

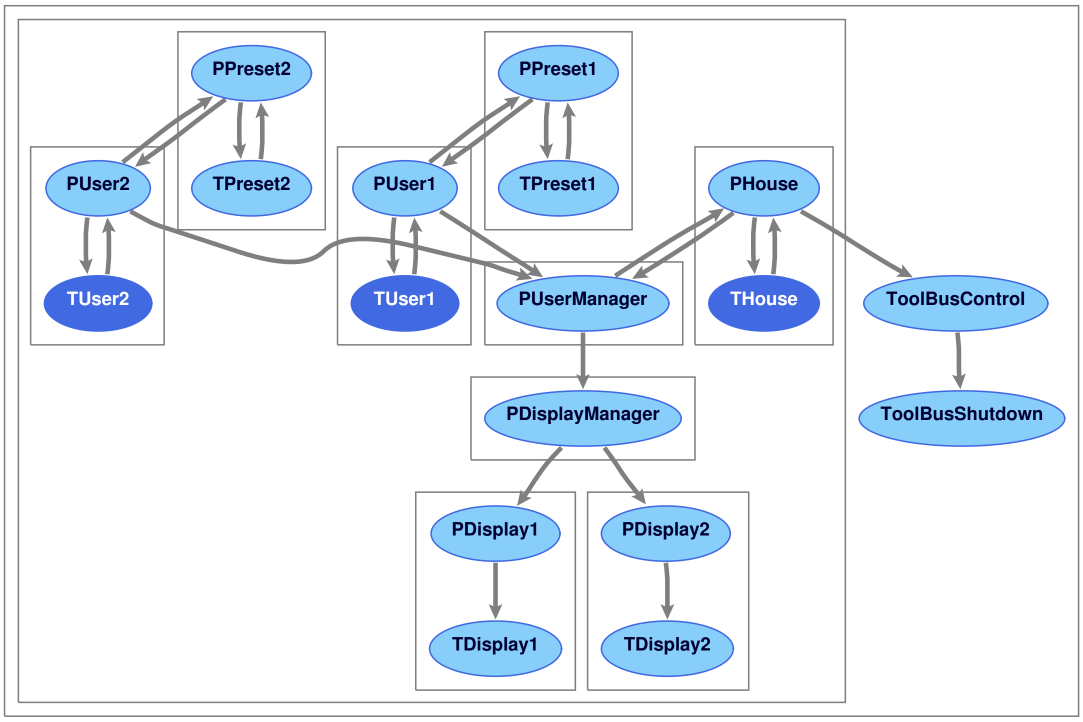


Figure 2.6: An animation of our completed Domotics application specification. Processes prefixed with the letter 'P' are ToolBus processes, those prefixed with 'T' represent tools.

processes are pure coordination processes, for example the UserManager or the DisplayManager. The programs provided with the software engineering pack for PSF do require all ToolBus processes to have tools, although this is by no means a requirement of the actual ToolBus. A simple workaround is to define tools for these processes as deadlock. Because the tool processes do not communicate the animator can be made not to display them.

2.5 Implementation

Now that we have an application specification the only thing left to do is to implement all the processes. As stated in Section 1.3.1 we can copy our PSF specification of the ToolBus processes almost verbatim into the T script. It must be noted however that porting our parametrized PSF specification requires considerably more work. Perhaps in the future a tool for automating this will be added to the software engineering pack for PSF.

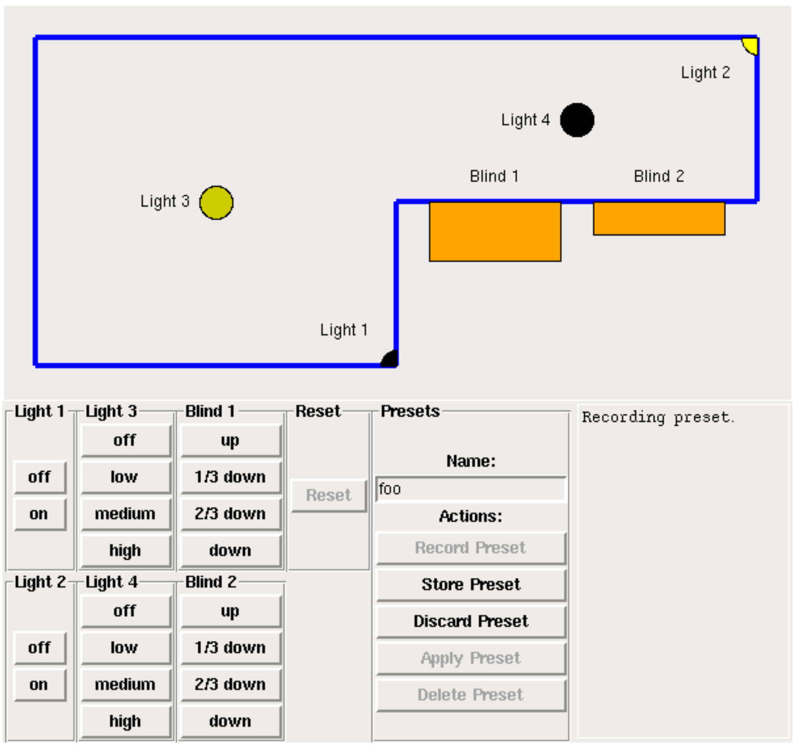
As stated before we have chosen to implement our tools in the Tcl/Tk language because of our need for a graphical user interface. Each physical user must be presented with instances of three different tools, a User, a Display and a Preset. To make the implementation more intuitive we have decided to present the users of our Domotics application with an aggregated GUI. An aggregated GUI allows multiple tools, each with their own GUI, to draw their interface components into a shared window. For example the PSF IDE [Die07] uses an aggregated GUI. In our aggregated GUI seen in Figure 2.7 interface elements from three different tools are present.

To show how there can be an infinite number of possible implementations for a given application specification we have parametrized the house. A single Tcl/Tk source file now contains several arrays describing each actuator, its name, how it should be drawn by the Display tools and so on. A possible extension to our Domotics application might be to write a program with a GUI that automatically generates this file.

The listing below shows some excerpts from the House tool of our Domotics application. When the tool is started the `Initialize` procedure is called on line 41. The `Initialize` procedure, lines 23 through 39, in turn draws several interface elements. A reset button, line 27, and a shutdown button, line 33. When the shutdown button is pressed the `Shutdown` procedure gets executed and the `quit` datum is sent, line 8. The `reset-datum` is sent on line 3 when the reset button is pressed. The buttons and procedures to simulate the hardware triggering of actuators are not shown in this listing.

The House tool also receives data, for example the `actuator-state` datum is received by the procedure on line 11 and the `reset-please` datum by that on line 15. All ToolBus tools must also have a procedure called `rec-terminate` to ensure tools are properly stopped when the ToolBus is shut down.

```
 1  proc Global-reset {} {
 2    [...]
 3    TBsend "snd-event(reset-please)"
 4    [...]
 5  }
 6
 7  proc Shutdown {} {
 8    TBsend "snd-event(quit)"
 9  }
10
11  proc actuator-state {actuator} {
12    [...]
13  }
14
15  proc reset-please {} {
16    [...]
17  }
18
19  proc rec-terminate {e} {
20    [...]
21  }
22
23  proc Initialize {} {
24    [...]
25
26    labelframe .reset -text "Reset"
27    button .reset.rst -text "Reset" -command {
28      Global-reset }
29    grid .reset -column 0 -row 0 -sticky nsew
30    grid .reset.rst
31
32    labelframe .shutdown -text "Shutdown"
33    button .shutdown.sht -text "Shutdown" -command {
34      Shutdown }
35    grid .shutdown -column 0 -row 1 -sticky nsew
36    grid .shutdown.sht
37
38    [...]
39  }
40
41  Initialize
```



```
Recording preset.
```

Figure 2.7: A screen capture of the aggregated GUI presented to users. The graphic of the house is rendered by the Display tool, the buttons below it are rendered by the User tool. In the bottom right a text field rendered by the Preset tool is visible.

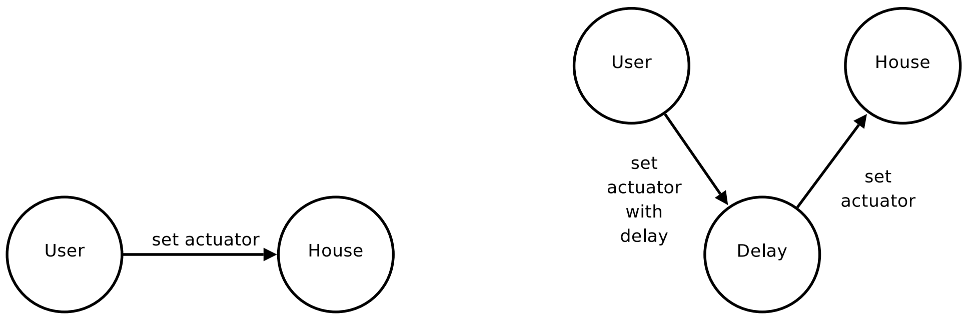
CHAPTER 3

Extending the Domotics application

To further examine the usefulness of the software engineering process with PSF we now propose several extensions to our Domotics application. When an application that has been engineered using the software engineering process with PSF is to be extended several steps must be taken. First the new requirements must be turned into scenarios, then the architecture specification is updated. When the architecture specification has been updated the application specification is next. After having updated the application specification we update the implementation, possibly adding new tools. All the while we ensure, using validation techniques, that the functionality of the pre-existing scenarios is not compromised.

3.1 Delayed setting of actuators

The first extension to our Domotics application we discuss is the ability to delay the setting of an actuator for a certain amount of time. This might for example allow a user to set his coffeemaker to turn on at 7 a.m. before going to bed. Figure 3.1 shows a diagram to clarify the semantics.



(a) Directly setting an actuator.

(b) Setting an actuator with a time delay.

Figure 3.1: A diagram showing the two methods of setting an actuator. An actuator can either be set directly or it can be set after a certain amount of time has passed. Note that the house cannot distinct between the two.

For extending our application we follow the same five steps as we did during the initial engineering process (see Section 1.4). We have already updated the requirements by stating that the system must also allow users to delay the setting of actuators.

We must now make the necessary modifications to our list of scenarios. In this case we opted to add two, delaying the setting of a single actuator and delaying the application of a preset. Deciding on these scenarios took a negligible amount of time.

Then we move on to our architecture specification. Since every actuator has only one instance, namely that represented in the house, it only seems logical to create a single, central, component to handle the delayed actuator settings. We call this component `timedelay` and connect it only to the UserManager. We now specify a new datum, `timed-actuator-state`, which represents the same information as the `actuator-state` but with an added element of time. Whenever a user wants to delay the setting of an actuator, he sends a `timed-actuator-state` to the UserManager. The UserManager in turn forwards this to the time delay component. When the time has come to set the actuator the time delay component sends an `actuator-state` to the UserManager, which then forwards it to the house and the displays as before. Figure 3.2 shows part of the animation for our updated architecture.

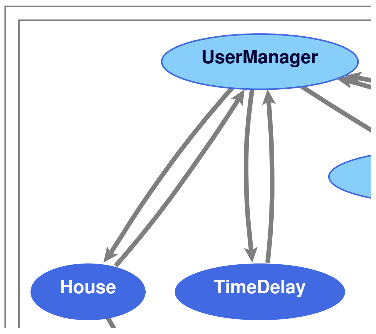


Figure 3.2: Part of an animation of the architecture specification for our Domotics application extended with the delayed setting of actuators. The part that is not visible is identical to the architecture in Figure 2.5.

We also need to allow presets to be applied after a certain delay. This requires only changes to the user and preset components. If a user wants to delay the application of a preset he communicates a `delay-apply-preset` datum to his preset component. The preset components will respond by sending zero or more `timed-actuator-state` data before sending the `preset-applied` datum.

These changes to our architecture specification took about 20 minutes to write and validate. It is immediately clear that this would not be possible had we written our specification as is done with traditional software engineering. Although the initial rewrite might have been done in a comparable amount of time, validating the changes would most surely have taken much longer.

Next we move on to our application specification. Because we only needed to specify one tool from scratch, the time delay tool, and make only minor changes to other tools, we were able to complete the rewrite of the application specification in about half an hour. This includes the time we took to validate the specification as described in Section 1.4.8. Because traditional software engineering does not have a phase comparable to an application specification it is hard to make a statement as to the relative efficiency of software engineering with PSF in this context.

Finally we updated our implementation, this took about 3.5 hours. That may seem long, but it is mainly caused by our lack of proficiency in the Tcl/Tk language. We dare to argue that an experienced Tcl/Tk programmer could have easily made and tested the changes in less then half the time it took us. Of these 3.5 hours we spent about half an hour updating our T script. This time could be significantly reduced if the PSF Toolkit contained a script or program to automatically convert our application specification into a T script.

3.2 Conditional setting of actuators

Another extension we implemented is the conditional setting of actuators. The implementation resembles that of the delayed setting of actuators. In this context we consider a condition to be that some actuator is in some state. For example a user might want the upstairs television turned off as soon as the downstairs one is turned on. Here we follow the exact same steps as with the delayed actuators.

First we add a scenario for the conditional setting of actuators. We then update our architecture specification to accommodate the new scenario of the conditional setting of actuators.

We add a `conditional-actuator-state` datum that represents the condition and the actuator to be set when the condition is met. We also add a component to handle the conditional setting of actuators, called `conditional`. When a user wants to set an actuator on a condition, the `user` component sends a `conditional-actuator-state` datum to the UserManager, which then forwards it to the `conditional` component. Because the `conditional` component needs to be aware when actuators change state, so it knows when a condition has been met, the UserManager also forwards `actuator-state` data to the `conditional` component. Figure 3.3 shows part of the animation for the updated architecture.

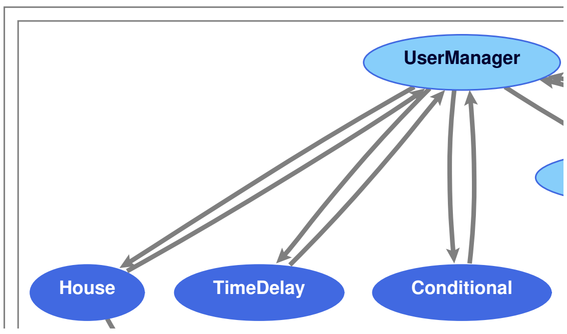


Figure 3.3: Part of an animation of the architecture specification for our Domotics application extended with the conditional setting of actuators. The part that is not visible is identical to the architecture in Figure 2.5.

We update our application specification and move on to the implementation. Figure 3.4 shows the final GUI of our Domotics application, including the delayed and conditional setting of actuators.

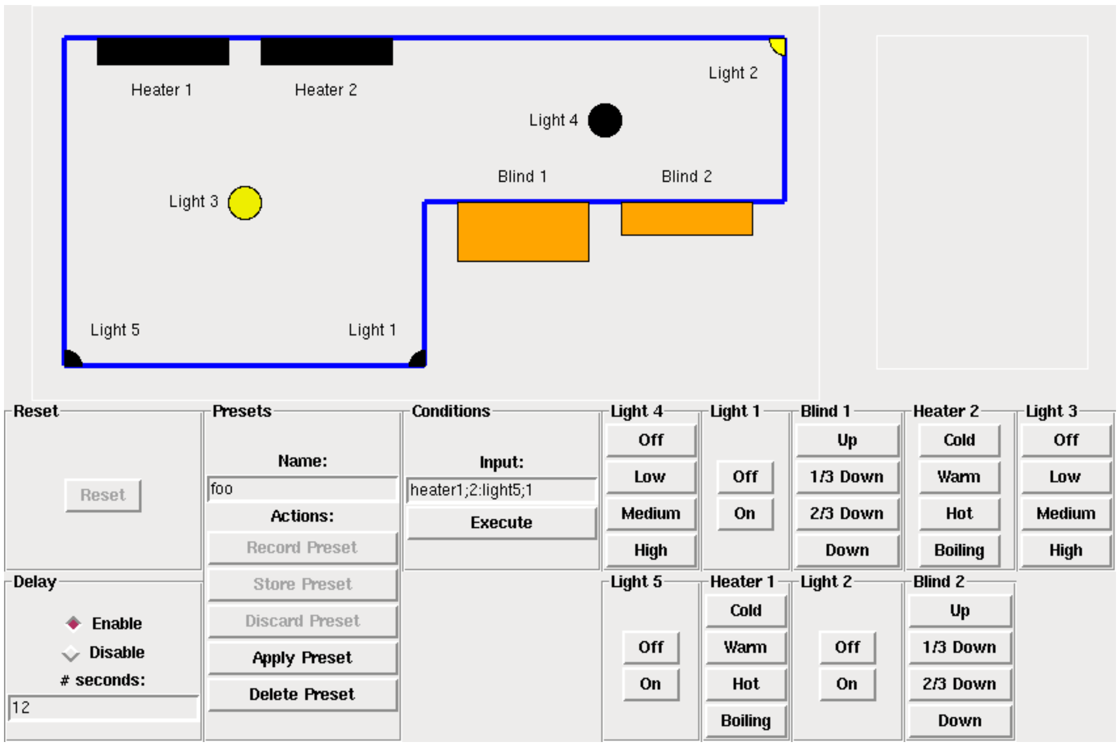
3.3 Monitoring power consumption

Yet another extension could be the monitoring of the power consumption of the house. We did not implement this extension, but we do discuss how this could be done. Monitoring power consumption forces us to make another assumption about the house, namely that it has power consumption monitoring equipment that is connected to the Domotics system. We would naturally begin by making a scenario for the monitoring of power consumption.

After that we would update the architecture specification. For this extension it is not necessary to add any new components to our architecture. Adding a datum describing the current power consumption would suffice. The implementation of the house, whether virtual or physical, must ensure that this information is sent to the Domotics system periodically. We would need to update the user manager components and the display components to handle the new datum.

If we have also implemented the conditional setting of actuators, the component responsible could also be updated, so that conditions can also include power consumption, e.g. turn off the air conditioner if the power consumption exceeds a certain amount. Note that in that case it would not be necessary to update the user components, because we have only specified the condition datum abstractly.

Updating the application specification would be very straightforward as would updating the implementation.



Name: foo

Input: heater1;2:light5;1

[✓ Enable] Disable

seconds: 12

Figure 3.4: A screen capture of our final aggregated GUI. The graphic of the house is rendered by the Display tool, the buttons below it are rendered by the User tool. In the top right a text field rendered by the Preset tool is visible.

CHAPTER 4

Conclusion

We have successfully engineered a Domotics application using the software engineering process with PSF. Successfully, because the final application meets all the requirements we imposed. While engineering the Domotics application we learned about the various stages of the software engineering process with PSF. We also learned how to use the PSF Toolkit, the software engineering library for PSF and the ToolBus.

Because we were able to validate our applications design during specification and continually feed back improvements to our specifications we ended up with a simple, yet effective architecture for our Domotics application. Because its design is split between a coordination architecture and tools it was easy to extend the Domotics application with the delayed setting of actuators.

Because the semantics for the more complex functionality, such as the creation of presets, was determined during specification we had a solid handle during implementation. After all we had already defined the behavior of each of our tools during the application specification phase. This proved particularly helpful because we had no experience programming in the Tcl/Tk language.

We conclude that the software engineering process has proved useful during the engineering of our Domotics application. In the following sections we discuss our practical experiences with the PSF Toolkit, the ToolBus and software engineering with PSF.

4.1 The PSF Toolkit

The PSF toolkit consisting of, amongst others, a simulator and an animator for PSF specifications provided us with the tools required to validate our specifications. Being able to see a representation of the various components of a system together with their interactions made it easy to maintain an overview of the system. This in turn lowered the barrier for us to make modifications to a specification when we found a flaw during implementation.

During the simulation and animation of the specifications for our Domotics application we did encounter some possibilities for improvements in the PSF Toolkit. For example when multiple processes have atoms with the same names the animator does not distinct between them, even though the simulator does and should. The result is that the animator will show some atoms in the choose list for the wrong process, possibly even indicating to the user that the process where the atoms were supposed to be displayed cannot execute any atoms at a certain point in time. We first encountered this when specifying the architecture for the third scenario, multiple users. We had multiple users, each with his own `push-set-actuator` atom, however they were all listed under one of the user components. The solution, or rather workaround, was to parametrize the atoms, so that the name of the component would become part of the atom, e.g. `push-set-actuator(user1)`.

Another welcome improvement to the animator would be the ability to visually 'collapse' constrained processes, e.g. collapse a pair of 'P' and 'T' processes to a single 'PT' process in an application specification. We encountered difficulties when we tried to simulate our final, parametrized application specification. To test whether our application would indeed function as expected with four users (we had previously tested it only with two) we animated the specification. Because there were now so many processes they would no longer all fit on the screen. The result was that we had to continually scroll the animators window up and down to see if our specification worked as expected.

The usefulness of the simulator might for example be improved by adding the ability to execute a trace of a previous execution. This would have allowed us to create traces of executions that resulted in deadlock. We would then be able to run the trace on the next version of our specification to see if the deadlock no longer occurred. Naturally that would require that the trace be applicable to the new specification, which need not be the case. We now had to manually execute all the atoms in a saved trace to see if we had resolved a deadlock.

4.2 The ToolBus

Because the current PSF software engineering process focuses on creating ToolBus applications, it stands to reason that we also discuss our experience with the ToolBus. The more complex an application becomes, the more levels of abstraction can be looked at. One of the most difficult tasks when engineering a complex application is dealing with the communication between and in general the coordination of the various components of the application. The ToolBus, by strictly separating the coordinating processes from the tools, conceptually simplifies the application as a whole by inviting the engineer to look at the application from a higher level of abstraction.

One of the major problems we encountered with the ToolBus was that the application library from the software engineering pack for PSF specified the ToolBus communication protocol rather abstractly. When we tried to convert our first application specification to an implementation the execution led to multiple protocol errors. We were at that time unaware of the difference between `snd-do()` and `snd-eval()`, see Section 1.3.2. Of course we should have more thoroughly researched the ToolBus communication protocol in the first place.

The ToolBus also has some features that are not specified, however abstractly, by the software engineering library. Although we had no particular need for any of those features there is one we believe is worth mentioning. The ToolBus supports the concept of 'notes', an instance of the publish/subscribe paradigm for asynchronous communication. For example we could have implemented our Domotics application in such a way that in the case of hardware triggered actuators the house would post the `actuator-state` datum to a message board containing notes, rather then sending it to the user manager. Anyone interested in the state of actuators, for example the displays, could then read the notes from the message board. Adding support for true asynchronous communication to the application library of the software engineering pack for PSF could potentially increase its usefulness for some types of applications.

4.3 The engineering process

We would first like to share an interesting observation we made while validating one of the final iterations of our architecture specification. We had unintentionally specified our architecture according to a rudimentary Model-View-Controller (MVC) paradigm [KP88], where the house was the model, the display the view and the user the controller. Because the MVC paradigm is widely used we believe this demonstrates, at least in our case, that we ended up with specifications that work on many levels of abstraction.

With regard to the tools of the software engineering pack for PSF (see Section 2.4) we would liked to have had a tool capable of converting a parametrized architecture specification to an application specification. This could easily be automated by extending the `se-constrain` tool to handle parametrized specifications.

Several additional tools might also make the life of the software engineer easier. Because PSF specifications are formally defined they can be converted into other formal languages. For certain languages verification tools have been written [GR01]. It may be valuable for engineers to be able to use these tools to prove their application possesses certain properties.

Bibliography

[BHK89] J.A. Bergstra, J. Heering, and P. Klint. The Algebraic Specification Formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. Addison-Wesley, 1989.

[BK87] J.A. Bergstra and J.W. Klop. ACP$_\tau$ — A Universal Axiom System for Process Specification. *CWI Quarterly*, 15:3–23, 1987.

[BK98] J.A. Bergstra and P. Klint. The discrete time ToolBus — A software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, July 1998.

[Die94] B. Diertens. New Features in PSF I: Interrupts, Disrupts and Priorities. Technical Report P9417, Programming Research Group — University of Amsterdam, June 1994.

[Die97] B. Diertens. Simulation and Animation of Process Algebra Specifications. Technical Report P9713, Programming Research Group – University of Amsterdam, September 1997.

[Die00] B. Diertens. Generation of Animations for Simulation of Process Algebra Specifications. Technical Report P0003, Programming Research Group – University of Amsterdam, October 2000.

[Die05] B. Diertens. Software (Re-)Engineering with PSF. Technical Report PRG0505, Programming Research Group – University of Amsterdam, October 2005.

[Die06] B. Diertens. Software (Re-)Engineering with PSF II: from architecture to implementation. Technical Report PRG0609, Programming Research Group – University of Amsterdam, November 2006.

[Die07] B. Diertens. Software (Re-)Engineering with PSF III: an IDE for PSF. Technical Report PRG0708, Programming Research Group – University of Amsterdam, October 2007.

[DP94] B. Diertens and A. Ponse. New Features in PSF II: Iteration and Nesting. Technical Report P9425, Programming Research Group — University of Amsterdam, October 1994.

[GR01] J.F. Groote and M.A. Reniers. Algebraic Process Verification. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 17, pages 1151–1208. Elsevier Science, March 2001.

[KP88] G.E. Krasner and S.T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.

[MV90] S. Mauw and G.J. Veltink. A Process Specification Formalism. *Fundamenta Informaticae*, 13(2):85–139, June 1990.

[MV93] S. Mauw and G.J. Veltink, editors. *Algebraic Specification of Communication Protocols*, volume 36 of *Cambridge Tracts In Theoretical Computer Science*. Cambridge University Press, 1993.

[Ous94] J.K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[Vel93] G.J. Veltink. The PSF toolkit. *Computer Networks and ISDN Systems*, 25(7):875–898, 1993.

[Vel95] G.J. Veltink. *Tools for PSF*. PhD thesis, Institute for logic, language and computation – University of Amsterdam, 1995.

[vW93] J.J. van Wamel. A library for PSF. Technical Report P9301, Programming Research Group — University of Amsterdam, 1993.