Managing Complexity in Modelling Embedded Systems

Saad Zafar
Institute for Integrated and Intelligent Systems, Griffith University, Australia

R. G. Dromey
Software Quality Institute, Griffith University, Australia

ABSTRACT

Increased dependence on embedded systems in daily life underlines the need for modelling techniques that are simple to use, and yet powerful enough to effectively manage the complexity and provide a platform for formal verification. The Genetic Software Engineering (GSE) method manages the complexity of embedded systems requirements by translating one requirement at a time into a graphic notation with a formal semantics called behavior trees (BT). Behavior trees may then be composed one at a time to create an integrated view of the system behaviour. A system design and a component-based architecture can then be systematically derived from the integrated view. The critical properties of the design can then be assured by model checking and theorem proving. To do this the BT specification is automatically translated into the formal specification language of Symbolic Analysis Laboratory (SAL) environment. The results of applying the GSE technique to the design and formal verification of the critical properties of an ambulatory infusion pump are used to illustrate the method.

Keywords: Formal Methods, Embedded System, Systems Engineering, Genetic Software Engineering, Behavior Trees

1. INTRODUCTION

The increased dependence of embedded systems in our daily life has not only increased the complexity of such systems but the consequences of failures are also becoming increasingly more serious. Computer based embedded systems are often responsible for critical functions in automobiles, aircraft and modern medical devices. These systems have to ensure availability of services and reliably maintain a number of safety conditions in the context of failure of the system in a variety of different ways (Perrow 1984; Leveson 1995; Neumann 1995).

Typically the design process of many software based systems is guided by standards based on the best practices of a particular industry (Herrmann 1999). However, it has been argued that due to the increasing complexity of modern systems, the design process should be augmented by better and more formal methods for providing assurance of critical system properties (McDermid and Kelly 2004). The use of formal methods has been limited due to their difficulty in comprehension and high cost and time that is typically associated with their use. The problem is compounded by the fact that in order to fully comprehend the required application and consequences of failure the complete system must be modelled, analysed and verified. Furthermore, the defects in requirements specifications and gaps in deriving software requirements from a system specification are implicated in many software failures (Knight 2002). Therefore, there is a need for techniques that not only support modelling the whole system but the formal verification of critical properties system while still being easy to use and understand (Lutz 2000; Knight 2002; McDermid 2002; Dunn 2003).

The GSE method (Dromey 2003) responds to this need by employing a simple graphical notation, which is easy to learn and use. The method manages the complexity of a system by translating system requirements into *requirements behavior trees* (RBT), one requirement at

a time. The RBT for each requirement is then integrated into an *integrated behavior tree* (IBT) in a manner analogous to putting pieces of jig-saw-puzzle together. The IBT is refined systematically into a *design behaviour tree* (DBT) by taking a number of design decisions. The impact of these design decisions is readily visualized as the changes are applied to an integrated view of the whole system. This reduces the information that needs to be kept in mind during requirements refinement and change. It is also possible to systematically derive the system architecture from the DBT which is presented in the form of component interaction network (CIN) diagram. The behaviour of individual components is projected out using component behaviour trees (CBT).

An important benefit of the method is its focus on early defect detection. Many incompleteness and ambiguities in requirements come into light during requirements translation. The requirements integration then helps us to detect and resolve other requirements integration problems associated with incompleteness and inconsistencies. The automated translation of a DBT into other formal specification languages like *Communicating Sequential Process* (CSP) (Hoare 1985) and *Symbolic Laboratory Analysis* (SAL) (Shankar 2000) then makes it possible to formally verify the critical system properties like safety conditions, liveness, deadlocks, etc.

In this paper we present results of applying the GSE method to design a medical device called an ambulatory infusion pump (AIP) (Deltec 2005). This safety critical device is used for patients that need measured doses of drugs infused at regular interval while away from direct medical care of health professionals. In the case study the informally stated requirements are first specified formally using the BT notation before they are integrated into an IBT. A number of defects are found during the translation and integration steps. The solution is then represented in a DBT which is systematically refined from IBT. As a last step we translate the DBT specification into SAL code for formal verification of system properties of interest.

The rest of the paper is organized as follows. Section 2 gives a brief overview of GSE and illustrates how a BT specification is translated into the SAL specification language. Section 3 presents the case study and illustrates how the AIP design is built out-of-its-requirements using the GSE approach. In section 4 we analyse the system design for its critical system properties and conclude our discussion in section 5, with a brief overview of related work and future directions for our research.

2. THE GENETIC SOFTWARE ENGINEERING DESIGN STRATEGY

The GSE method differs from conventional design strategies by aiming to build system *out-of-its-requirements* rather than aiming to build system that just *satisfy* a set of given requirements (Dromey 2003). The translation of requirements into BT specifications is a fundamental concept for achieving this GSE objective. During the translation process the focus is on the requirement being translated and not the system as a whole. The premise is that dealing with manageable chunks of information at one time allows us to operate without stretching the limits of our short term memory. In these circumstances we are better able to deal with detail without making mistakes.

A behavior tree node is used to specify a component and its state. It is also used to express conditions, events, attribute assignment and data-in and data-out behaviours of a component. Two BT nodes are joined together with an arrowed line representing causal behaviour and the direction of flow of control. Two nodes joined together with a line without an arrow specify an atomic action with causal dependency. Atomic actions without causal dependencies are specified as BT nodes boxed together without a joining line between them (see figure 1, for examples).

The GSE method seeks to capture and preserve the intent of each individual requirement by translating the informally stated requirements sentence-by-sentence and word-by-word. Any

implied behaviour or missing behaviour is clearly marked to indicate deviation from the given requirements. A "+" sign is used to mark implied behaviour and "-" sign is used to mark missing behaviour. The traceability between the requirements and their BT representation is maintained by the use of traceability tags in the notation which references the corresponding requirement being translated. As an example, consider the statement, "when the user presses the button then the timer is started". Figure 2 illustrates how this requirement is translated into a BT representation. A BT representation of a single requirement statement is commonly referred to as requirements behavior tree (RBT).

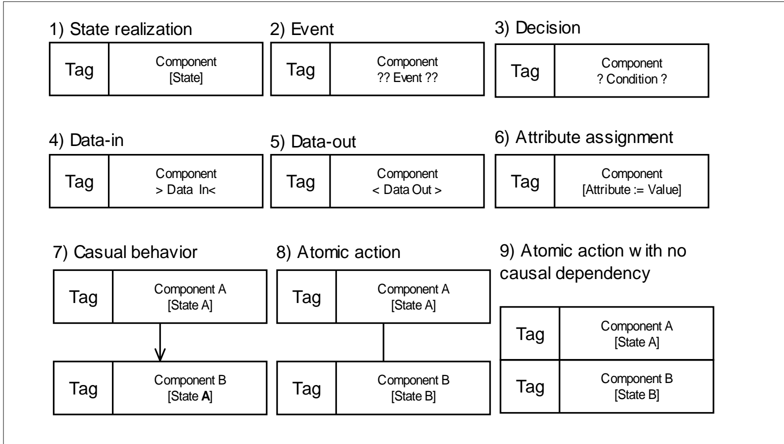

Figure 1: A summary of BT notation syntax

As discussed earlier, the concept of integrating RBTs into a single IBT is similar to putting pieces of a jig-saw-puzzle. However, the RBTs will only perfectly integrate if all the pre- and post-conditions have been clearly defined and there are no incompleteness in the requirements. Therefore, the integration step plays an important role in resolving requirements problems like incompleteness and inconsistency of requirements. Without resolving these issues the RBTs will simply not integrate. As one RBT establishes the pre-condition that the other RBT needs. The integration techniques along with some integration axioms for integration are presented in (Dromey 2005). The system architecture is derived from the DBT and is represented in the form of component interaction network (CIN). It is also possible to project from the DBT the component behaviour designs of all the components in the system. These are represented as component behavior trees (CBT).

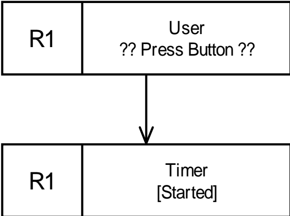

Figure 2: Translation of a requirement to a behaviour tree

The static analysis of GSE models is possible by translating the BT specification into SAL specification language. SAL is an integrated environment of static analysis tools that include tools for model checking and theorem proving (Shankar 2000). In the SAL environment the systems are specified using a description language for state transition. The system properties of interest are calculated from SAL based on the system expressed as a transition system in this description language. In the SAL environment a number of tools provide support for abstraction, program analysis, theorem proving, and, model checking.

A detailed description of the translation of BT to SAL specification is beyond the scope of this paper but a brief overview has been provided here. Since, in BT the concurrent systems are

expressed as state transitions, the translation rules for a subset of BT notation is relatively straight forward. A wider coverage of translation of the BT notation is part of the ongoing research (ARC 2004). A BT is represented in a single SAL transition system module. The components and their states are declared as state types in the module. The BT events that are marked with INPUT tag are translated as input variables. A set of special variables called pc1..pcN (program counters) is used to keep track of concurrent state transitions in the tree. An atomic action can be either manually specified or automatically generated from set of BT state transitions between two external (observable) events (Grunske, Lindsay et al. 2005).

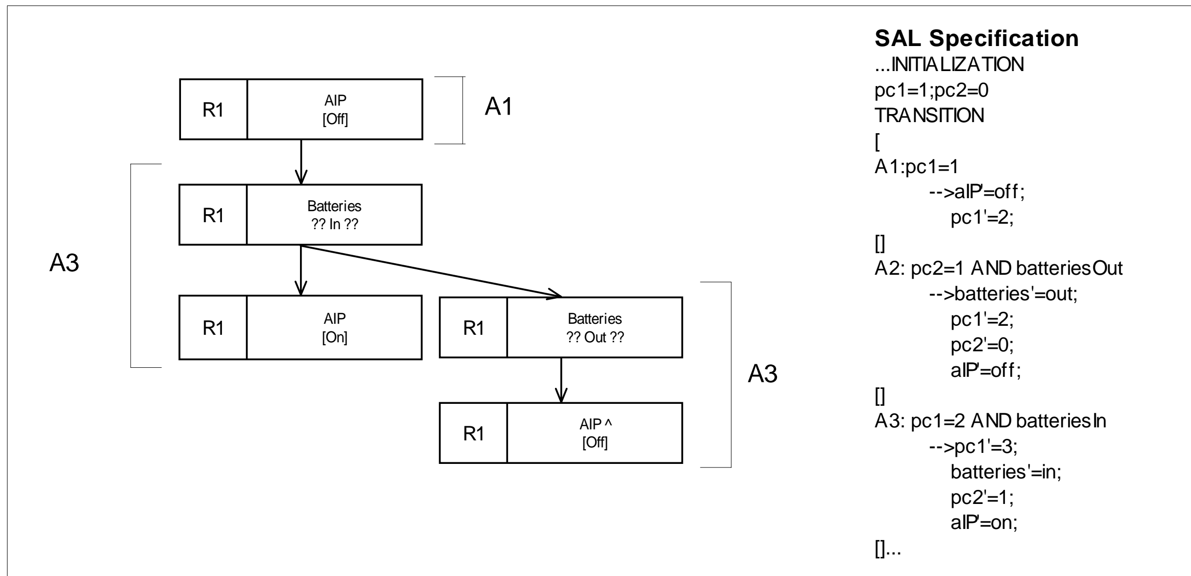

Figure 3: Translation of BT notation into SAL specification

Consider the BT fragment and its SAL translation illustrated in figure 3. In the SAL specification the root node transition is performed in action labelled A1. This action is guarded by program counter pc1=1. Since the program counter has been initialized to 1, therefore, the control flow starts from this action. After the state transition the program counter pc1 is set to 2. The action 'A3' is guarded by the condition pc1=2 and by an input variable 'batteriesIn', which corresponds to BT event 'Batteries ?? In ??'. When both of the conditions are true the program counter is incremented to pc1=3 and the AIP state is changed from aIP=off to aIP=on. At this stage a concurrent thread is also spawned, which waits for the 'Batteries ?? Out ??' event. In the SAL specification this is handled by setting the other program counter (pc2) to 1. Therefore, when the pc2 is set to 1 and the input variable 'batteriesOut' is true then the action 'A2' is performed. This action causes the system to revert back to the root node in the behavior tree, i.e. AIP [Off]. The reversion in BT specification is specified by the "^" symbol. This reversion effect is translated in the SAL specification by setting the system states and program counter states to the match the state of action A1. Now the system is once again ready to transition into action A3 when the input variable 'batteriesIn' becomes true.

Once the system has been specified in the SAL environment language, a number of analyses can be performed on the system specification (Moura 2004). The sal-sim tool is a SAL simulator which is used to show different execution paths. The evaluation of different execution paths is usually a desirable first step to assess the correctness of the specification. Similarly, another tool, sal-path-finder produces a random trace. The SAL specification can also be checked for deadlocks using dead-lock-checker tool. Useful properties of the system can be specified using linear temporal logic (LTL) or computation tree logic (CTL). These properties can then be model checked using sal-smc and sal-bmc tools.

A Behaviour Tree Editor (BTE) allows us to draw and edit BT (Smith, Winter et al. 2004) in a graphical environment. The tool stores the BT specification in XML format. In addition to the

facility of translating BT specification into SAL code, the tool also supports translation of BTs into the CSP representation (Winter 2004). The FDR model-checker is then used for model checking the CSP specification to verify the BT model. We will now show how these design and verification steps are applied to the ambulatory infusion pump system.

3. AMBULATORY INFUSION PUMP – A CASE STUDY

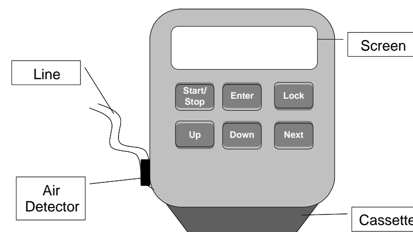

Figure 4: Ambulatory Infusion Pump

An ambulatory infusion pump (AIP) is a modern safety critical device that is used to deliver measured doses of drug to patients while they are away from direct care of health professionals. The device is programmable to allow health professionals to configure it to meet patient's drug therapy requirements. AIP has an inbuilt pump which delivers the drug to the patient based on the programmed infusion rate. The device connects to the patients through a line with a needle assembly (see Figure 4).

Table 1: AIP requirements

No.	Requirements
R1.	The system is turned on when the batteries are put in and is turned off when the batteries are out.
R2.	To start the pump, when in *stopped* state, *start-stop* button is held down until it beeps three times and (… … …) is displayed on the screen.
R3.	To stop the pump, when in *running* state, *start-stop* button is held down until it beeps three times and (… … …) is displayed on the screen.
R4.	When the battery is low, the system sends three beeps and displays *battery low* message on the screen.
R5.	Every time the system pumps 1 ml of drug when the battery is low it sends a single beep alarm.
R6.	After a set time pump activates to pump 1ml of drug through the line.
R7.	When the volume reaches 5ml the system does three beeps and displays *volume low* message every 1ml as it counts down to empty.
R8.	When there is no drug left, the pump enters *stopped* mode and the system sounds a continuous beeping alarm.
R9.	When the line is closed/blocked or kinked the system does a constant alarm beeping if it is in the *running* mode.
R10.	Whenever air is detected in the line, by the air detector sensor, the pump is stopped and the beeper sounds a continuous beep.
R11.	The *main* screen displays the pump status (*running/stopped*), battery status (*normal/low*) and drug *volume*.
R12.	If no key is pressed for 2 minutes then the screen is reset to the main screen

Due to the ambulatory nature of the device and criticality of the drug therapy there are certain hazards associated with the device. For some drugs used in the therapy it is critical that drug is infused into the patient at regular intervals. Under-delivery or non-delivery of the drug may lead to death or may cause serious injury to the patient. This hazard can arise from the line being blocked or kinked while the pump is running. Therefore, it is critical that the AIP system is able to detect such a blockage in the line and take an appropriate measure to avoid any serious consequences. Similarly, drug over-delivery can occur due to programming error.

This is possible in a situation when there is a discrepancy in the actual drug delivered and the calculation of the amount of drug delivered by the AIP controller software.

Another hazard to consider is presence of air bubbles in the line while the drug is being infused. This could lead to a medical condition called air embolism which could also result in serious injury or death of the patient. Other hazards associated with the infusion pump could result from unauthorized programming of the device are not addressed in this paper.

The requirements for AIP in this case study have been derived from the user manual of a commercial product CADD-Legacy ® Ambulatory Infusion Pump (Deltec 2005). The subset of requirements relevant to this paper is given in the table 1. The pump's programming functions that allow clinical staff to program in the drug therapy setting have been excluded from the scope of this paper to make the system and development process easy to understand.

3.1 Requirements Translation

As a first step in the GSE process, the given requirements are translated into a BT specification one at a time. Figure 5 and 6 illustrate the requirements translation method. In both the examples the BT is developed right out of the given requirements. All the assumptions during the translation are clearly marked. For instance, for requirement statement R2 a number of assumptions had to be made: 1) it is the user (operator) who presses the button, 2) a timer component (Timer {Beeper}) monitors the time delay equivalent to beeper beeping three times, 3) when the user releases button before the beeper has finished beeping the beeper must be stopped (along with the timer) and the system must revert back to the stopped state, 4) if the beeper has beeped three times (i.e. the timer signals that it has waited as long as 'beep delay', the system goes into 'AIP [Pump [Running]]' state. All the other requirements are translated one by one in a similar manner.

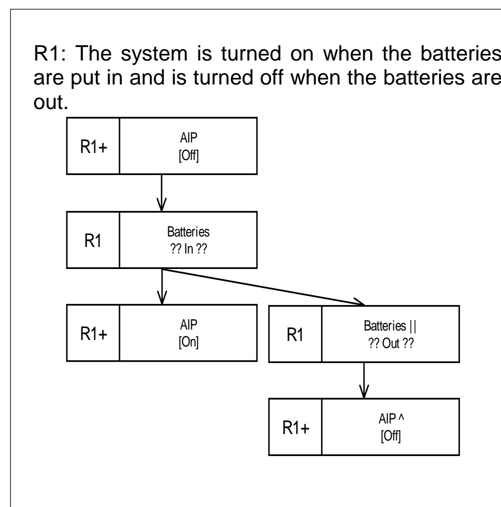

Figure 5: RBT for requirement R1

The outcome of this first step of requirements translation is that all the given requirements have been formally specified in BT notation and the intent of each of the individual requirement have been preserved as RBTs. The traceability between requirements and their formal representation has been maintained by marking the traceability tag with corresponding requirement number. The assumptions for any implied or missing behaviour have also been clearly marked.

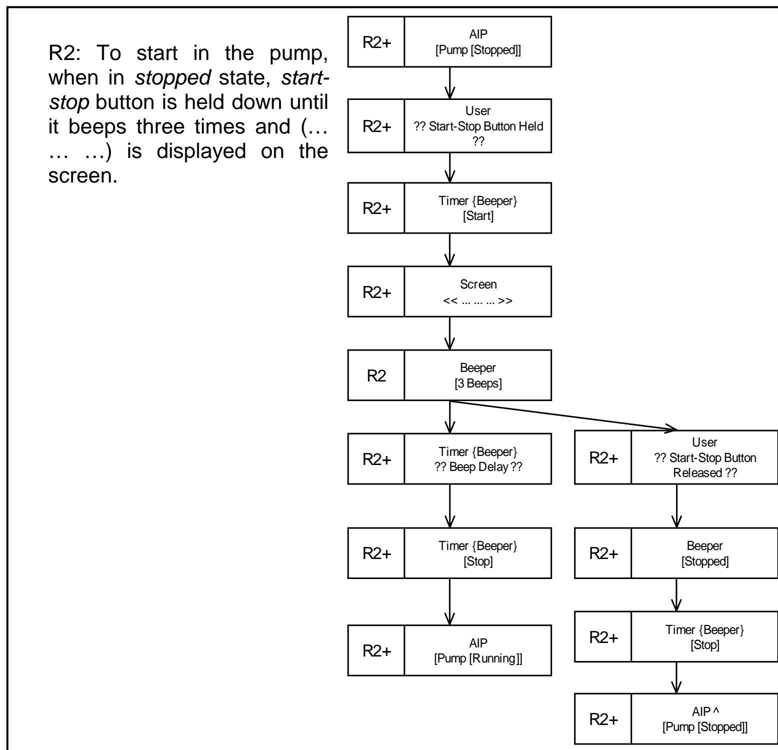

Figure 6: RBT for Requirement R2

3.2 Integrating Requirements

Once all the requirements are translated into corresponding RBTs, the next step is to integrate them. In the AIP system, the pre-conditions of the each requirement statement have not been clearly defined. For instance, the requirement statement R4 simply states that "when the batteries are low, the system sends three beeps". This statement does not specify when the AIP system should implement this behaviour. This integration problem forces us to make an assumption that the system should monitor the batteries voltage as soon as it is turned on. Once again, this assumption is also marked and should be validated. Figure 7, illustrates the integration of R4 into the IBT. The other RBTs reveal similar problems during this step of requirements integration.

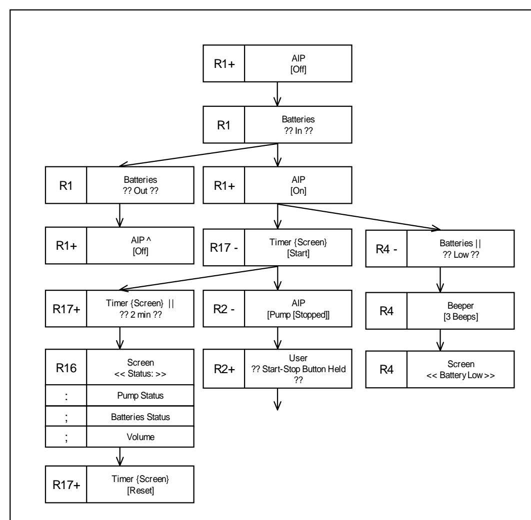

Figure 7: Integration of R4 into an IBT

In addition to the identification of defects in requirements, the IBT plays another important role in systems development. It provides an integrated view of the specified requirements, which is then used to derive the system's design in a systematic manner. In this way the

impact of each design decision on the originally specified system is readily visualized, thereby, providing a systematic platform for transitioning from problem domain to solution domain in the form of a DBT. More importantly, the complexity of design process is reduced by reducing the amount of information that has to be kept in mind at one time.

3.3 Developing a Design Behaviour Tree

The GSE method facilitates design decisions by providing an integrated view of the whole system while developing DBT. Some simple guidelines along with domain knowledge may be used for refinement of the IBT into a DBT. The discussion on the refinement is beyond the scope of this paper but several examples are provided here.

As an example, consider the BT segment on left hand side in figure 8. The causal behaviour in this BT segment states that user's holding the start-stop button will cause the Timer {Screen} to be reset. This causal behaviour may be refined as user holding the button will cause the (physical) Start-Stop Button to be held, which would in turn cause the internal state of AIP controller button to be set to held 'AIP [Button [Held]]'.

In a similar manner other refinement techniques are used to systematically derive the system design. These techniques include decoupling operator, sensor and actuator behaviours. This separation of concern helps in defining system boundaries and also assists in failure mode analysis of various components for safety critical systems. The causal behaviour in the DBT is further refined by identifying messages to asynchronous components which branch out of the normal flow of BT as leaf nodes. For instance, in the AIP system, all the messages to the beeper component are asynchronous.

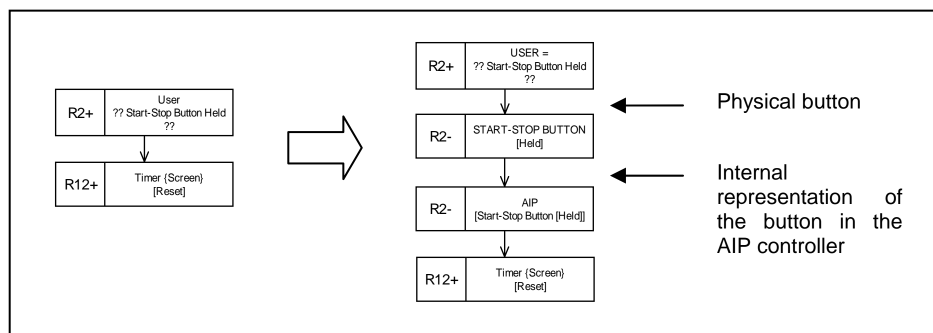

Figure 8: Refining causal behaviour in DBT

Component transitions with causal dependencies in an atomic action are specified using lines with no arrow, representing no time between the state transitions. The atomic action where there is not causal dependency or where the dependency does not matter is specified by boxing the BT nodes together without drawing any line between them. In a safety critical system, such as this one, interrupt handling and protecting critical regions is an important consideration. We discuss these and other safety concerns later in the next section.

In the AIP case study we were able to identify some incomplete critical behaviour during the refinement process. Some of the important findings from this step are summarized in the table 2. Perhaps, the most important is the refinement of requirement R7. The requirement states that pump should count down the amount of drug infused by 1ml every time it pumps the equivalent amount of drug. This pump operation can be interrupted by the user pressing the start-stop button or the AIP controller aborting the pump operation if there is air in the line or blockage of the line has been detected by the system. At this stage the pump might be in the middle of pumping the drug. It is therefore, important to re-calculate the amount of drug given so far. If the pump's pumping rate is known, then it might be possible to calculate the amount as a function of time since the pump started pumping the drug. As we will discuss later, excluding this behaviour could lead to a new system hazard of drug overdose.

The complete integrated and refined DBT of AIP system is shown in figure 9. Due to the lack of space we cannot present a readable design of the system. The figure is intended to give the reader a feel of size and structure of the final DBT.

Table 2: Requirements refinement of AIP system

No.	Description
R1	There is an initialization state in AIP controller, called AIP [Initialize], before it is turned. In this state the controller sets up threads for monitoring of sensors and buttons behaviour.
R4, R5	There is a voltage detector that detects charge in batteries and sends the signal 'normal' if the batteries are normal or sends batteries 'low' message otherwise. The signals from voltage detector sets the internal states AIP [Batteries [Normal]] and AIP [Batteries [Low]], respectively.
R2, R3, R8, R9, R10	Displaying critical state changes on the screen: - 'Running' when the pump goes into running state - 'Stopped' when the pump goes into stopped state - 'Line Blocked' when the blockage in the line has been detected - 'Air in Line' when the air in the line has been detected - 'Empty' when the system runs out of drug
R11	The status section of the screen is updated when: - pump is stopped - pump is set to running mode - batteries go low - every time the volume of the drug is changed
R7	The drug remaining is re-calculated (as a function of time passed since the last pump time) whenever the pump operation is interrupted.

3.4 Translation of DBT into SAL

To support the static analysis of our design we have translated the DBT specification into SAL specification using the BTE tool. Due to limitations of the tool and the syntactical requirements of SAL some deviations from the original DBT specification had to be made. For instance, the state names in the DBT were made unique to meet the SAL requirements. Similarly, due to tool limitations, all the asynchronous messages, shown in the DBT as separate branches, had to be changed to be included in the same branch to suggest that that state change in the component occurs at the same time the message is sent. However, these changes do not impact the overall design of DBT. Similarly, the selection of the atomic action by the tool had to be manually verified. The translated code was model checked for deadlocks and traces were produced to analyse the accuracy of the specification. The results from this step were satisfactory. Model checking the AIP design for its safety properties is discussed in the next section.

4. ANALYSING THE AIP DESIGN

The AIP system presented in this paper is a safety critical device. The failure of correct function of the pump might lead to death or serious injury to the patient. The hazards in the system include, drug under delivery, drug over dose and air embolism. These hazards may occur due to malfunction of the device or unauthorized or wrong programming of the pump. As pointed out earlier, in this paper we only consider hazards associated with the malfunction of the device. Malfunction could be in the form of failure of sensors or due to programming errors in the controller software.

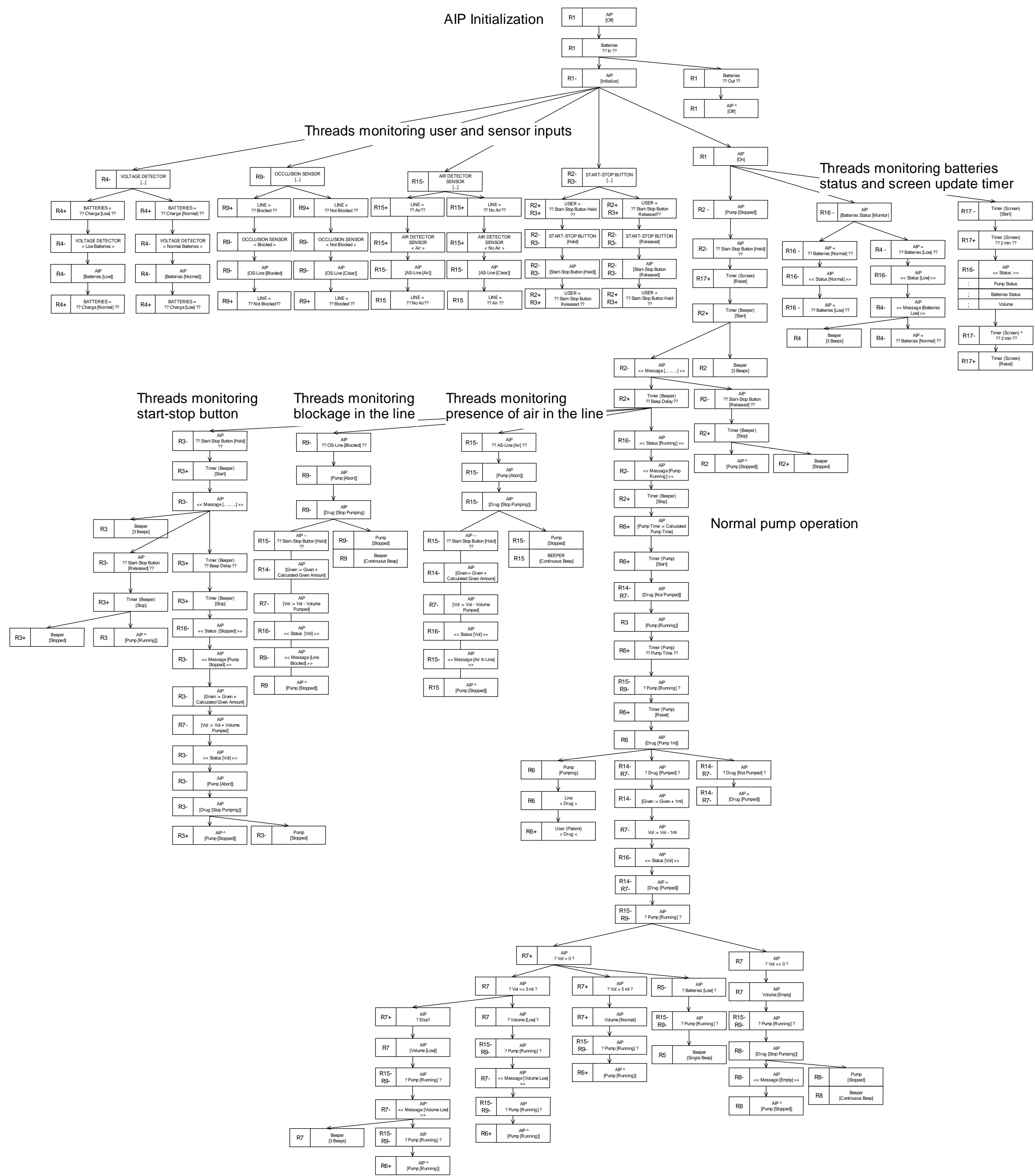

Figure 9: The integrated and refined DBT for AIP system

The drug under delivery hazard may occur if blockage in the line is not detected. This may be due to failure of the occlusion sensor. Similarly, air embolism hazard may occur due to the failure of air detector sensor. On the other hand, wrong calculation of drug infused may be due a software error, which may lead to drug overdose or drug under delivery hazard. Table 3 summarizes these hazards and the potential causes and some of the failure modes that may lead to the identified hazards.

All of these hazards may occur during the pumping operation. In the AIP design this pumping operation occurs in a loop which is controlled by a timer component named ‘Timer {Pump}’. The timer activates the pump every time it counts up to ‘pump-time’. The calculation of value for ‘pump-time’ is based on the programmed infusion rate of the pump. At ‘pump-time’ the controller sends a signal to the pump to infuse another 1ml of drug. Once the drug has been

pumped 1ml is subtracted from the amount of drug volume. At this stage the controller performs a number of checks and warns the users if the remaining drug level is low or has reached the empty state. The user is also warned if the batteries are low. These warnings are both in the form of displayed messages on the screen and beeping sounds from the beeper.

Table 3: AIP system hazards

No.	Hazard	Cause	Failure modes
H1	Under-delivery or non-delivery of drug	- Obstruction or kinks in the line - Incorrect calculation drug infused	- Occlusion sensor failure - Programming error
H2	Air embolism	- Presence of air in the line	- Air detector sensor failure
H3	Over delivery of drug	- Incorrect calculation of drug infused	- Programming error

In our design, in parallel to the pump operation described above, we have specified three threads that wait for; 1) air-detector sensor to sense air in the line, 2) the occlusion sensor senses blockage in the line, and 3) start-stop button to be held (figure 10). The first two threads immediately interrupt the pump operation and the last thread only interrupts the pump operation if the button is held long enough. The specification of interrupt handling in BT notation is discussed in the next section.

4.1. Interrupt Handling

In most safety critical systems it is vitally important to interrupt the normal behaviour of the system whenever the system senses some potential hazardous situation. In such cases timely interruption of the normal behaviour in a graceful manner is critical. The behaviour tree notation provides semantics for thread interruption, which are described here in the context of our case study.

Figure 10 illustrates interruption of the normal pump operation (thread 4) by thread 1 (start-stop button held long enough), thread 2 (occlusion sensor detects line blockage) and thread 3 (air detector senses air in the line). To interrupt the pump operation, the interrupting threads simply change the state of ‘AIP [Pump [Running]]’ to ‘AIP [Pump [Abort]]’ and in the thread itself guards are placed before all the critical actions to check if the pump operation should continue or not. For instance when the ‘Timer {Pump}’ indicates it is time to pump another 1ml of dose then at that very instance the state of ‘AIP [Pump [Running]]’ might be changed by any of the interrupting threads. If this is the case the control of flow would be ceased and the thread will die off.

Similarly, this guard is placed in other places in the thread running the normal pump operation. For instance, consider an interrupt by line blockage (thread 2). If the line blockage is detected then it will, among other things, send a stop signal to the pump, display the ‘line blocked’ message on the screen, sound a continuous signal and revert back to the stopped state. But if in between this time the thread running the pump operation detects that the drug volume is zero then it will also stop the pump and send a continuous beep. However, the message displayed this time would be ‘empty’. Therefore, it is important to place the guard before the ‘empty’ message is displayed to ensure that wrong message is not displayed in this situation. In our design all the message and beep alarms are protected with a guard to avoid any misleading information being displayed on the screen.

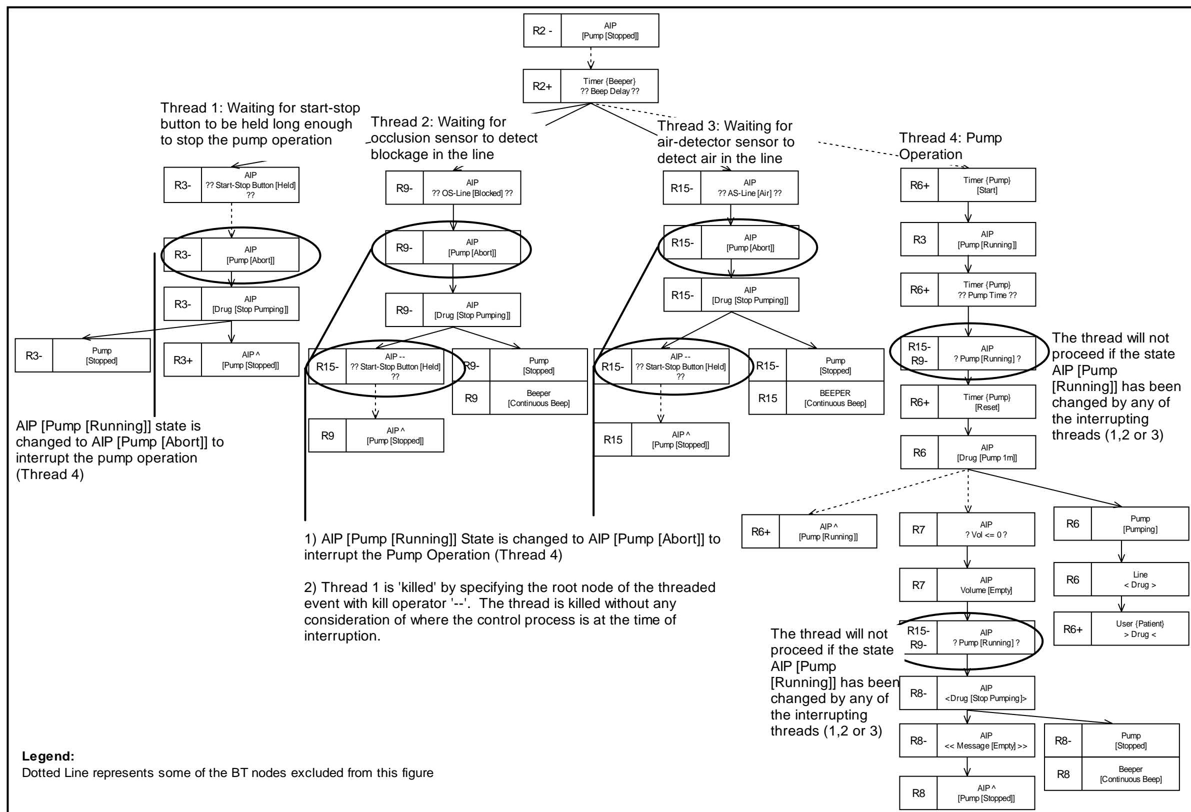

Figure 10: Specifying thread interruption in BT

Another way to specify killing of a thread is by using the kill operator '--'. In our example (figure 10) thread 1 is killed in this manner by thread 2 and 3. In case of thread 1 we do not really care where the flow of control is at the moment the thread is interrupted. If the line blockage or air in the line has been detected, the thread should just simply be killed.

4.2. Verifying System Properties

An important step in development of safety critical system is to provide assurance that the system's safety properties are not violated. The automated translation of the DBT specification into SAL specification language help us model check safety condition of the AIP system. For the purpose of this case study we have identified the following conditions:

1. If there is air detected in the line then the pump must not pump drug to the patient.
2. If blockage of the line is detected then the pump must not pump drug to the patient.
3. If the pump operation is interrupted then the drug volume must be re-calculated.
4. If the drug volume is zero then the pump must not pump drug to the patient.

The failure of first safety condition may lead to the hazard of air embolism. The failure of the second and third condition may lead to drug under/non-delivery and over-delivery respectively. While the combination of fourth and first condition may also lead to the hazard of air embolism. The SAL tools sal-smc and sal-bmc were used to verify these safety conditions. The safety conditions were expressed in LTL formulae in the SAL environment. The results of the model checking are illustrated in table 4.

The LTL formula for the first safety condition (th1) states that it is globally true (represented by 'G' operator) whenever there is air in the line (line=air) the pump should be stopped (pump=pStopped) in the next step (represented by 'X' operator). The formula for the second condition (th2) is similar to th1. Both the theorems were proved. The third condition, which states that it is globally true that whenever the system sends the signal to the pump (aIP__Drug=stopPumping), the volume (aIP_vol) is re-calculated. This ensures that the

actual amount of the drug matches attribute value of the volume within the controller component.

Table 4: Results of model checking safety properties of AIP system

No	Condition	Ref.	LTL Formula	Outcome
1.	If there is air in the line the pump should not pump	Th1.	G((line=air) => (X(pump=pStopped)))	Proved
2.	If there is blockage of the line the pump should not pump	Th2.	G((line=blocked) => (X (pump=pStopped))	Proved
3.	If the pump is stopped then the drug volume must be re-calculated	Th3.	G((aIP__Drug=stopPumping) => (aIP__Vol=vCalculated))	Proved
4.	If the drug volume is zero then pump must not pump the drug	Th4.	G((pump=running) => NOT (aIP__Volume = empty))	Not Proved

The last condition demonstrates a flaw in our design. This safety condition is derived from requirement statement R8. The statement requires the pump to be stopped once there is no drug left to be pumped. But it is not explicitly stated that the controller should not allow the pump to restart if the volume is still zero. In our design we have not put any guard which disallows pump operation if the volume is zero. Violation of this safety condition alone does not cause direct harm but it is only in combination of failure of the first condition it may cause the hazard of air embolism. We were only able to uncover this design error during the formal verification of our design. This underlines the importance of using formal methods for design verification.

5. CONCLUSION

The GSE approach to systems development aims control the complexity of designing modern embedded systems by reducing the amount of information that needs to be processed at a given time. During requirements translation this means that the complexity is reduced to consideration of one requirement at a time and during the design phase all the impact of all the design decisions are readily visualized, thereby reducing the load on our short term memory. The simple graphical form of the BT with its formal semantics is also intended to enhance the understandability of the model. The automated translation of the BT specification into SAL enables us to formally model-check the critical properties of the system.

The case study presented in this paper illustrates how the GSE method is employed to design and then formally verify the safety critical properties of the system. In the case study we have shown how defects found early in the requirements specification may be documented and suggest how all the assumption may be validated. Furthermore, each of the given requirements is traceable to the final design. The requirements refinement was performed in a systematic manner in the form of evolving the DBT. During the refinement process the impact of each design decision was apparent when the DBT was modified. The fact we were able to uncover design flaws in our DBT through model checking reinforces the idea of using formal methods in design of critical applications.

The limitations of the GSE approach include lack of support for timing and performance analysis, which is the subject of current research work at (ARC 2004). Furthermore, GSE model transformation into UML and automated code generation are also being investigated. To support the GSE design process, a hazards and failure mode analysis technique is being developed. Integration of safety and security requirements into design of complex system using the GSE approach is also part of our ongoing research.

Acknowledgement: This work is partially funded by Australian Research Council (ARC) under the ARC Centres of Excellence program. The authors wish to thank their colleagues in the Dependable Complex Computer-based Systems project for their constructive suggestions. We would also like to acknowledge the contribution of Dr. Lars Grunske and

Nisansala Yatapanage for their assistance in model checking our design using SAL specification.

6. REFERENCES

ARC (2004). ARC Project, Building Dependability into Complex Computer Based Systems, ARC Centre for Complex Systems.

Deltec (2005). Operator's Manual, CADD-Legacy ® 1 Ambulatory Infusion Pump, Model 6400, Smiths Medical MD, Inc. **2005**.

Dromey, R. G. (2003). From Requirements to Design: Formalizing the Key Steps. (Invited Keynote Address) SEFM-2003, IEEE International Conference on Software Engineering and Formal Methods, Brisbane.

Dromey, R. G. (2005). "Genetic Design: Amplifying Our Ability to Deal With Requirements Complexity." Lecture Notes in Computer Science **3466**: 95-108.

Dunn, W. R. (2003). "Designing safety-critical computer systems." Computer **36**(11): 40-46.

Grunske, L., P. Lindsay, et al. (2005). An Automated Failure Mode and Effect Analysis based on High-Level Design Specification with Behavior Trees. Fifth International Conference on Integrated Formal Methods (IFM2005), Eindhoven, The Netherlands.

Herrmann, D. S. (1999). Software Safety and Reliability: Techniques, Approaches, and Standards of Key Industrial Sectors, IEEE Computer Society Press.

Hoare, C. A. R. (1985). Communicating sequential processes, Prentice Hall International - Englewood Cliffs, N.J.

Knight, J. C. (2002). Safety Critical Systems: Challenges and Directions. 24th International Conference on Software Engineering, ICSE 2002, Orlando, Florida.

Leveson, N. G. (1995). Safeware: System Safety and Computers, Addison-Wesley Publishing Company.

Lutz, R. (2000). Software engineering for safety: A roadmap. The Future of Software Engineering, ACM Press, New York.

McDermid, J. (2002). Software Hazard and Safety Analysis. Formal Techniques in Real-Time and Fault-Tolerant Systems, 7th International Symposium, FTRTFT 2002, Oldenburg, Germany, Springer-Verlag.

McDermid, J. and T. Kelly (2004). Software in Safety Critical Systems: Achievement and Predictions. 5th International Conference on Control and Instrumentation in Nuclear Installation, Institute of Nuclear Engineers.

Moura, L. d. (2004). SAL: Tutorial, SRI International.

Neumann, P. (1995). Computer related risks, ACM Press.

Perrow, C. (1984). Normal accidents: living with high-risk technologies. New York, Basic Books.

Shankar, N. (2000). Combining Theorem Proving and Model Checking through Symbolic Analysis. CONCUR'00: Concurrency Theory, Springer-Verlag.

Smith, C., K. Winter, et al. (2004). An Environment for Building a System Out of Its Requirements. Tools Track, 19th IEEE International Conference on Automated Software Engineering, Linz, Austria.

Winter, K. (2004). Formalising Behavior Trees with CSP. International Conference on Integrated Formal Methods, IFM'04, LNCS.