Runtime Validation for Embedded Real-Time Systems

A Validation Method for Real-Time Systems in Simulated Environments

Anders B. Christensen
abc@cs.auc.dk

Master Thesis

Department of Computer Science
Aalborg University

January 17th 2003

Aalborg University
Department of Computer Science

Title:
Runtime Validation for Embedded Real-Time Systems – A Validation Method for Real-Time Systems in Simulated Environments

Semester:
Fall 2002, Dat6

Project Period:
September 1st 2002 to
January 17th 2003

Author:
Anders B. Christensen

Supervisor:
Kim G. Larsen

Number of Pages:
82

Copies:
3

Keywords:
Real-Time Systems, Validation, Real-Time Specification for Java, Runtime Verification, Simulation, Embedded Systems

ABSTRACT:

With the increasing use of computer systems for controlling processes and activities in our everyday surroundings, the correctness of these systems becomes increasingly important. The functionality of such control systems often relies on tasks being performed at the correct time, not to soon and not to late.

In this report we investigate the contemporary practices for ensuring correct behaviour of such *real-time systems*. We identify the quality and applicability measures of current practices as well as their limitations. Based on the contemporary methods, we propose a method for validating embedded real-time systems.

The proposed method allows real-time systems to be validated in simulated environments. The simulation performed is based on formal descriptions (models) of the environments. During simulation, the structure of the model is used by a guiding algorithm to achieve an interesting exploration of the possible behaviour. Furthermore, validation is performed in terms of the states of the environment.

A prototype of the proposed method has been implemented. Based on this prototype we carry out a case study illustrating the process of describing and formalising the environment as well as validating programs controlling it.

Preface

This report documents the Master Thesis by *Anders B. Christensen* (05-04-77-2187), written under the research unit of *Distributed Systems and Semantics* at the *Department of Computer Science* at Aalborg University. The project is concerned with the validation of real-time systems and proposes a validation method for such systems.

The author would like to thank Professor *Kim Guldstrand Larsen* for his enthusiastic and inspiring supervision throughout the project. Furthermore, the author wishes to thank Doctor *Klaus Havelund* (who holds a research position at *Kestrel Technology*, *NASA Ames Research Center*, *Moffet Field Air Base*, *California*), and his colleagues for their aid in obtaining an overview of current practices within model checking and runtime verification.

Anders B. Christensen

Contents

Introduction CHAPTER 1

Today, computer systems are heavily used in fields requiring control of one or more processes. In a typical scenario such *control programs* continuously observe the entities in its surroundings and generate appropriate reactions. Consider, for example, a simple system controlling the temperature of a cooling facility. By reading a sensor value, the system may monitor the current temperature. When a reading shows that the temperature has exceeded five degrees Celsius, an appropriate reaction may be to switch on a cooling device. If the temperature was to ever rise above eight degrees Celsius an alarm could be switched on.

Often control systems can be classified as *critical*, that is, erroneous behaviour of the control logic used may result in disastrous situations, e.g. the loss of human life. Naturally, ensuring that such errors do not occur is an important discipline. However, it also represents a complicated and tedious task – more often than not accounting for a significant amount of the total effort spent in a project.

An important class of control systems is that of *real-time systems*. Systems in this class are characterised by the fact that their overall correctness depends not only on the correctness of results produced, but also on the *correct timing* of these results. For example, in the cooling facility example given above, it may be imperative that the cooling device is switched on within some known interval of time, in order to attain a low fluctuation in the temperature level. Thus, even though the cooler device is eventually switched on, the system is considered to function correctly only if the cooler was indeed switched on within the required interval of time.

In this project we continue the work presented in [Chr02], where a framework for validation of real-time systems was introduced. In this framework, named the *Inquisitor Framework*, it is possible to execute a real-time system in parallel with a simulation of the environment in which it resides. For example, with the example presented above, the control system would interact with a set of processes representing the environment. One process would represent the cooling device which can be turned on and off. A second process could represent the current temperature of the cooling facility – increasing and decreasing over time depending on the behaviour of the control systems and an assumption about the temperature of the surroundings.

For the validation purpose, a third set of processes known as *inquisitors*, is introduced. These processes have the responsibility of *monitoring* the execution of the control system. The behaviour of the control system is then *checked* for accordance with properties defined by a person performing the test. For example, in the cooling facility example, one such property is that the temperature of the system never rises above eight degrees Celsius. Furthermore, the inquisition processes may be used to *guide* the behaviour of certain aspects of the environment. We shall elaborate on this functionality in a later chapter.

As described above, a set of environment simulation processes and a set of inquisition processes is required in order to perform validation in the Inquisitor Framework. The application logic of these processes must be supplied by the team of people testing a control system. It is desirable to minimise the overhead imposed by having to implement environment and inquisition processes in order to lower the required effort for testing. It is equally important to obtain a systematic approach for validation in order to ensure a thorough testing of the control system.

The work presented in this report aims at achieving these goals. For this purpose we introduce concepts used in formal validation methods and adapt them for use in this practical setting. Specifically we use the formalism of automata theory to represent the behaviour of the environment. The processes needed to simulate the environment are generated from these models, and the inquisition processes use the structure to guide the behaviour of the environment.

In this chapter we shall first describe how control systems interact with their surroundings. We then elaborate on the conditions for validating real-time systems and define the goals and quality measures for validation. This is followed by an introduction to three contemporary validation methods, *model checking*, *runtime validation*, and *model-based testing*. We analyse their strengths and weaknesses according to the goals and measures defined. We also discuss how the interaction with the surrounding environment is accomplished in the three methods. Towards the end of the chapter, we propose a validation method based on runtime validation, and state the goals and scope of this project. Finally, we give an overview of the remainder of the report.

1.1 Real-Time Systems and their Environments

As earlier mentioned, real-time systems are often used to control some process in their physical surrounding. A system is typically designed to control a specific environment – in fact, it is common for real-time systems to be *embedded* within some physical entity. For example, in a microwave oven, an embedded control program can be used to react to events in the surroundings – such as the push of a button.

In such *embedded systems*, the control program is executed on a hardware platform which we shall refer to as an *embedded device*. These typically consist of a processor, some memory, and a number of communication ports. Many embedded systems are consumer products (microwave ovens, television sets, etc.), which ship in large quantities. Considerable cost reductions can therefore be achieved by using embedded devices with minimum processing and memory resources. As a consequence, the binding of the control program to its environment is further tightened by the use of custom hardware configurations.

In order to interact with the entities of the environment, a conversion from the physical representation to a data-oriented representation is required. For example, in the cooling facility example, it is necessary to convert the temperature of the surroundings to a numeric value in order to monitor the temperature. Similarly, when the control program needs to start the cooler, a numeric value (or *signal*) must be converted to a switch in a relay. Figure 1.1 illustrates the setting of the cooling facility control program.

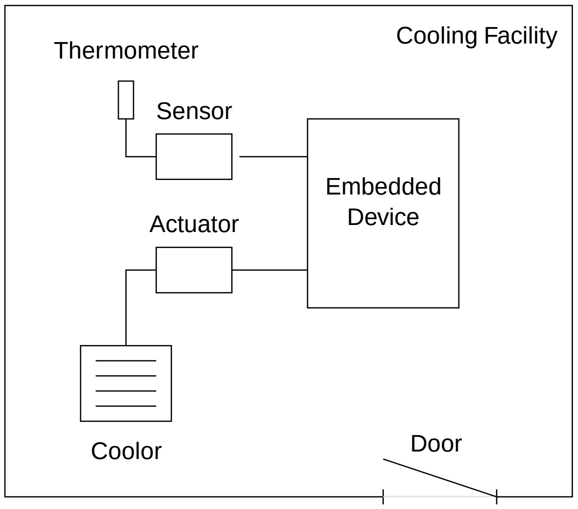

Figure 1.1: The cooling facility example. The control program executing on the embedded device is interacting with a thermometer and a cooler through sensors and actuators, respectively.

The conversions mentioned above are carried out by electrical and mechanical devices, known as *sensors* and *actuators*. The relationship between the embedded device, its sensor and actuators, and the physical environment is illustrated in figure 1.2. The sensors and actuators are connected to the communication ports of the embedded device. By reading the values of the input ports, the control program obtains knowledge about the current state of its environment, and by sending signals to the output ports it affects the future state of the environment.

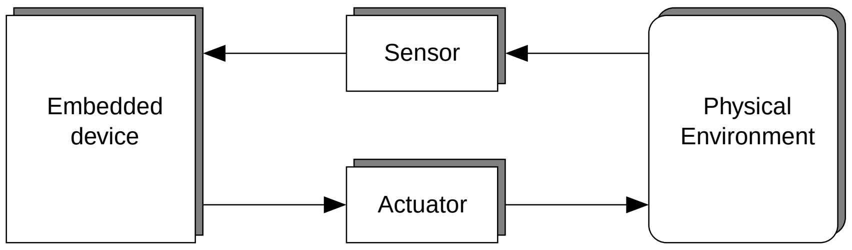

Figure 1.2: Embedded systems interact with their environments through sensor and actuator devices.

1.2 Validation Conditions

In this section we elaborate on the conditions under which development and validation of real-time systems are performed. Some of these motivated the implementation of the

Inquisitor Framework – others are dealt with by the validation method proposed later in this chapter.

- *High reliability and robustness demands* – Since real-time systems are often used for controlling some task or process in its surrounding, the reliability and robustness demands imposed on such systems are high. For example, the consequences of a break-down in a control program at a nuclear facility could result in grave human and economic losses.

  Another characteristic or real-time systems is that they are often part of *embedded* devices used in everyday appliances such as cars, microwave-ovens, and washing machines. Such devices are often produced in large quantities, leading to significant costs if an entire product line must be retracted due to an error;

- *Platform dependence* – With current practices, applications are often tightly bound to the underlying hardware platform. That is, specialised functionality provided by the platform are used at a low level of abstraction. As a consequence, switching the hardware on which applications run may require significant effort for redesigning and porting.

  This dependence on a given hardware platform makes developers vulnerable to change in hardware specifications and to the discontinuation of product lines. Furthermore, reuse of code from component libraries is complicated. In addition to the extended effort needed to recode functionality, quality of the software produced may also suffer due to programmers becoming sluggish because of a lack of intellectual interest in rewriting code.

  Hence, an abstraction from hardware is desirable from a quality as well as an economic view. One way to introduce such an abstraction is by the use of a *virtual machine* – which executes so-called *byte-code*. This byte-code may then be executed on every platform for which an appropriate virtual machine exists.

  An attempt to achieve this for real-time systems is made by the introduction of the Real-Time Java Specification. The Inquisitor Framework is based on this specification and thereby serves as a validation tool for programs written in compliance with it;

- *Well-defined environments* – Due to the high reliability and robustness demands, a considerable amount of effort is often spent defining the environments in which they reside. Having a good understanding of the behaviour of the environment is vital to produce control programs that respond correctly to given situations. These representations of the environment can be formalised and be used in the validation process. As we shall later describe, this is utilised in the case of model checking (see section 1.4.1).

- *Difficult testing conditions* – In some cases, testing software in the environment in which it is to be deployed is not possible. For example, for large systems, interacting with a lot of physical components, it may be too expensive to have an installation at the development site. Thus, integration tests cannot be performed until the end of the development process – which may lead to a late discovery of errors;

- *Analysis of real-time requirements* – For real-time systems, an important activity is that of estimating the *worst-case execution time* of processes. These calculations are used in an analysis showing whether or not the real-time requirements of the system can be met on a given platform.

  Estimating worst-case execution times is a complex and often time consuming process. One way of obtaining a measure of the worst-case execution time is by executing the application and measuring the time a task actually takes. However, when there are several threads in the application – and these threads are not *independent*, that is, they share some resource, such measuring can become very complicated. For example, if two processes both require exclusive access to some shared resource, the worst-case execution of one thread should not include the time spent waiting for access to a resource held by another thread.

  Another approach is to analyse the system using knowledge about the execution platform. This, however, is complicated by the complex designs of modern processors, due to the heavy usage of pipelines, caches etc. Also, if changes to the requirement specification are common, so will the need to recalculate worst-case execution times be;

- *Complicated debugging of real-time systems* – When an error is found during development or testing of a real-time system, finding the cause of the error is complicated by the existence of timing requirements. For example, the activity of *"step-through debugging"* often cannot be used as this drastically changes the timing of the system.

Some of these problems, notably the issue of platform-dependence are dealt with by the Inquisitor Framework. Since most of the issues are directly related to validation, we now turn to an investigation of the goals and measures of validation. This includes an investigation of the quality of software validation processes and some relevant practices currently in use.

1.3 Validation Goals and Measures

The goal of all validation is a very concrete one: To find as many errors as possible in a given program. In a traditional scenario, testing is carried out by defining *test cases* and writing *test drivers* that execute a program (or part of it) with some input and records output for analysis. Certainly, such test cases should ensure that the program is thoroughly tested. In other words, the quality of the test is only as good as is the quality of the test cases. For validation methods in general, we shall define the following three measures for the quality of a particular method:

- *Coverage* – As argued above, some measure for how thoroughly a program is checked is desirable. To this end we introduce the concept of *coverage* – denoting how large a part of the program behaviour has been subject to validation. One unit in which coverage can be measured is executed lines of code. If, after validation, 80 percent of the total amount of lines of code have been examined, by executing that line at least once, we shall say that the coverage is 80 percent.

A simple measure such as code-line coverage is often insufficient for calculating the coverage of program behaviour. Consider, for example, a function taking an integer parameter that is required to be in an interval between 0 and 4. If this function is called with the parameter value 1 it may be the case that all code lines of that function are covered and that the test is successful. This is, however, no guarantee that the function does in fact not fail on input value 4. In such cases, more abstract measures of coverage, often based on program *state*, are necessary. We investigate the coverage measures of some relevant current validation practices in section 1.4;

- *Types of errors that can be discovered* – Software errors take many forms. Some are rather abstract like the usability of a system or its conformance to a requirement specification. Other errors are closer tied to the code produced – for example deadlocks, race conditions, and various memory related errors. Errors in the latter category typically manifest themselves as system malfunctions or even break-downs. As will be described in section 1.4, different approaches operate on different abstraction levels of a system and therefore are able to reveal different types of errors;

- *False negatives and positives* – Say that a program is checked for a supported type of error using some validation approach. Expected behaviour of the process would be to report an error if and only if one was present. However, validation methods may sometimes falsely conclude that an error is present, even though this is not the case, or conversely, fail to detect an error that is indeed present. We shall refer to these undesirable phenomena as *false positives* and *false negatives*, respectively.

  False positives may lead to developers fixing errors that are, in fact, not present – inducing an overhead on validation effort. False negatives are an indication of insufficiencies of the validation approach. Thus, if a given approach is prone to false negatives it cannot, in general, be concluded that the program does not contain an error even though no evidence was found that it does.

Though a validation approach produces high quality results it may still not be very *applicable* in real-world validation practices. The applicability of validation approaches are characterised by the following properties:

- *Automation* – An ideal validation method is one of total automation – taking a representation of program as input and reporting errors found as output without the involvement of manual decision making or labour. At the other end of the scale, validation approaches without the aid of tools depend solely on manual labour and therefore require large amounts of effort for complex system. In addition, manual validation approaches are inherently more prone to errors. In conclusion, a high level of automation is desirable – a very low one may easily render an approach useless;

- *Scalability* – In order to meet the large and complex systems encountered in the software industry, a validation method needs to scale well;

- *Level of abstraction* – Validation methods differ in the level of abstraction at which they operate on application logic. For example, the traditional testing approach described earlier operates directly on compiled applications by executing it using test data. Other methods, including static analysis, operate on the source code without actually executing it, while yet others operate on high-level abstractions such as *models*. Depending on the level of abstraction, validation approaches may be applied at different stages in the development process. Naturally, as the level of abstraction increases, the types of errors that can be found change from implementation errors, such as memory and null-pointer references, towards high-level specification errors related to the design of the system.

1.4 Current Practices

Having identified validation of real-time systems as an important and complicated task, we now turn our attention to a review of some current validation practices. An overview of some current practices is given in figure 1.3. Testing approaches such as unit and integration testing are commonly used in validating real-life real-time systems. These approaches tend to be rather informal in nature. At the other end of the scale, formal verification methods (notably model checking, which shall be further described later) have proven successful – as tools such as Spin ([Hol97]) and Uppaal ([LPY97]) suggest.

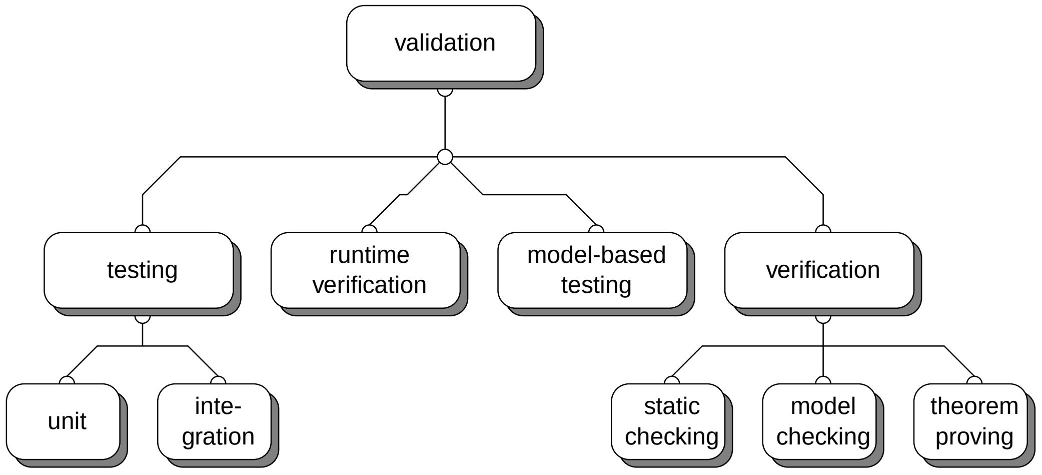

Figure 1.3: Validation approaches for real-time systems

Between the formal and the informal methods, some hybrids between the two extremes exist. Runtime verification is based on a single executing of a program and validates that this *trace* has certain properties. Model based testing uses *specification models* for describing allowed input/output behaviours of a system. Using the specification model, an implementation is then tested for *conformance* by feeding it allowed input and checking output produced for validity according to the specification model.

In this section, we shall introduce the validation methods of model checking, runtime verification, and model-based testing. These methods are considered relevant to our proposed validation method, described in section 1.5. For each approach we give a short description of a typical validation process and investigate the quality and applicability characteristics stated in the previous section. An overview is given in table 1.1, which also serves as grounds for a description of the limitations in current practices given in section 1.4.4.

1.4.1 Model Checking

The model checking process can be divided into two parts: A design part where a *model* of the system is created and a validation part where the model is *checked* either by *simulation* or *verification*. For the construction of the model the software designers identify abstract *states* in the system and *transitions* between them.

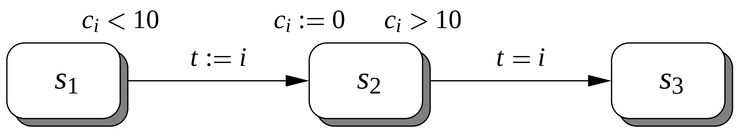

Figure 1.4: A single process in Fischer's protocol for mutual exclusion.

For real-time systems, a popular notion is that of *Timed Automata* (introduced in [AD94]), in which *clocks* can be defined and clock values may be used as restrictions on transitions and states. As an example, we consider Fischer's protocol for mutual exclusion. Each process in the protocol is an instance of the Timed Automaton depicted in figure 1.4, and includes a private clock c_i (initialised to zero) and access to the shared integer variable t which acts as a token. Each process must leave state s_1 before ten time units – thereby updating the value of the token. Upon entering state s_2 process i resets its clock to zero and remains in state s_2 until at least ten time units have progressed – ensuring all processes have entered s_2. Finally, having ensured that the token will no longer be updated, the process identified by the token will enter s_3, obtaining exclusive access to some resource.

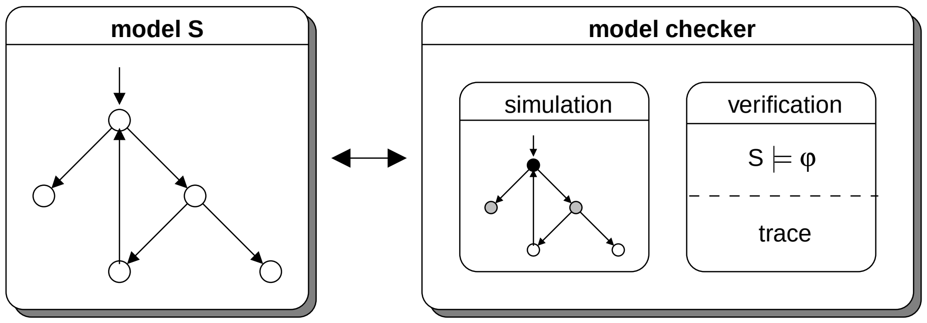

Figure 1.5: The model checking process

	Model Checking	**Runtime Verification**	**Model-based Testing**
Abstraction	Exploration of high-level models of applications in terms of *states* and *transitions*. There is a so-called *semantic gap* between the model and its implementation.	Based on the monitoring of a single *run* of modified application code.	Exploration of an implementation based on a specification model (transition system) defining allowed input and output behaviour.
Coverage	Based on abstract program state, full *state space* exploration ensures complete coverage.	Coverage is measured in the percentage of lines of code and may vary significantly.	Measured in terms of code lines as well as specification model state.
Error Types	Based on validation of properties. Due to the high abstraction level the error types supported are design-level errors including violation of *Safety* and *Liveness* properties.	Based on verification of properties. The program is monitored for violations during execution. Some properties such as deadlock, data-races, and runtime exceptions can be checked implicitly.	No properties except the specification model need be supplied. Detectable errors include errors where algorithms produce erroneous results and runtime errors.
False Pos/Neg	Since models are over-abstractions, false negatives can occur.	Subject to false positives as well as negatives.	Subject to false positives (like testing).
Automation	The modelling process is (most commonly) manual – however often this model can be used for automatic generation of control logic. Properties must be manually stated.	The modification of source code and monitoring of the execution is performed automatically. Some properties must be explicitly stated while others are checked implicitly.	The model must be provided manually. Testing is then performed automatically and may easily be repeated on different or altered implementations
Scalability	Subject to *state space explosion* imposing limitations to the size of models checked.	Good scalability.	Good scalability

Table 1.1: The quality and applicability properties of the validation approaches considered.

Validation

For the model checking part, tools such as Uppaal ([LPY97]) and Kronos ([Yov97]) are often used. Validation is performed by simulation and verification (illustrated in figure 1.5). In the simulation process, the behaviour of the system may be observed by interactively choosing the next state to enter when several possibilities exist. This activity is useful for design and debugging purposes.

An automated validation approach is that of *state exploration*, offered by the verification process. During this process, the system is examined and it is decided whether some given property holds or not. If not, a *trace* describing the error is provided. In the case of Fischer's protocol, properties may include that at most one process is in state s_3 at a time and that a process will eventually obtain exclusive access. These properties are instances of two categories of properties known as *safety* and *liveness*, respectively. Often informally explained as *"something bad never happens"* and *"something good eventually happens"*, such properties constitute prototypical *design-level* errors that can be discovered using model checking.

When a model checker performs a state exploration, each state encountered is checked for the property. If the property does not hold, an error trace is given. Otherwise, the state is stored – representing the fact that the given state is known to satisfy the property. Hereby, coverage, measured in states, is complete and false positives and negatives do not occur – presuming a correct model of the real-time system.

However, since time is continuous, the *state space* becomes infinite. This is clearly undesirable and techniques such as *symbolic model checking* are applied in order to restrict the space consumed. Still, model checking is subject to the phenomenon of *state space explosion*, where the model checker runs out of memory due to a large amount of stored states. This imposes a low scalability on the model checking technique.

Quality and Applicability

The fact that model checking is performed on a model rather than on the actual system limits the properties that may be checked to the design-level properties. This distance between the model and the actual system is often referred to as a *semantic gap*. Due to this gap it cannot be verified that an *implementation* of the system is correct – even though the model has been shown to be. Consequently, the high abstraction level of model checking imposes limitations on the types of errors that can be found.

The manual parts of model checking includes model construction, defining properties to be verified, and interaction during simulation. The most significant of these tasks is constructing a model of the system to validate. However, this task may be seen as part of the design process and therefore serves purposes exceeding validation of the system. For example, it is common practice to generate control logic for applications from models within the field of embedded systems engineering.

Finally, it should be mentioned that recent work has been conducted which narrows in the gap between models and actual code by generating models from the source code of programs. This work is most notably represented by the Bandera tool-kit ([CDH$^+$00], [DH99], and [DHJ$^+$00]), which contains a front-end for generating models from Java code and a back-end for interfacing with model checkers such as Spin ([Hol97]) and Java Pathfinder ([HP00]). Generating Promela models of software written in C++ (for verification with Spin) is supported by the FeaVer tool ([Hol00]). At the time of writing, none of these approaches incorporate timing requirements.

Representation of the Environment

When validating real-time systems using contemporary model checking tools, the environment is represented by models similar to those representing the control logic. Communication between these entities can be obtained in two ways:

- *Shared variables.* It is possible to declare global variables whose values can be read and reassigned from the environment as well as the control logic models;
- *Synchronisation channels.* In model checking tools, such as Uppaal, it is possible to declare channels on which synchronisation can be performed. The synchronisation process is illustrated in figure 1.6. During validation, two models can synchronise if both of the following hold:
  + The current state of one model allows a transition to be taken which can emit an *output* on a given synchronisation channel;
  + The current state of the other model allows a transition to be taken which can accept an *input* on the same synchronisation channel.

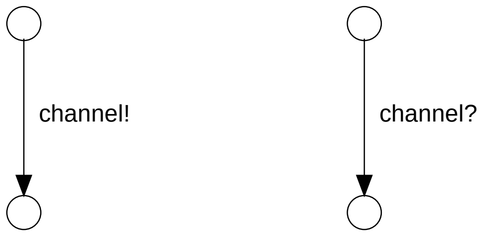

Figure 1.6: Two models synchronising during validation

In formalisms which contain the notion of time, the synchronisation action occurs at the same instant in both automata. Specifically, this is the case for Timed Automata.

1.4.2 Runtime Verification

Whereas model checking operates on abstract models of systems *runtime validation*, recently by [HP00] and [KKL$^+$01], is performed directly on the program by executing it and observing whether or not the program behaves as expected. In order to perform this monitoring, programs are modified to emit output describing certain events. The result of a *run* of the program is therefore a finite *trace* of events emitted by the program during execution. The expected behaviour is given by one or more properties, and the system is considered to be well-behaved if a trace satisfies a given property.

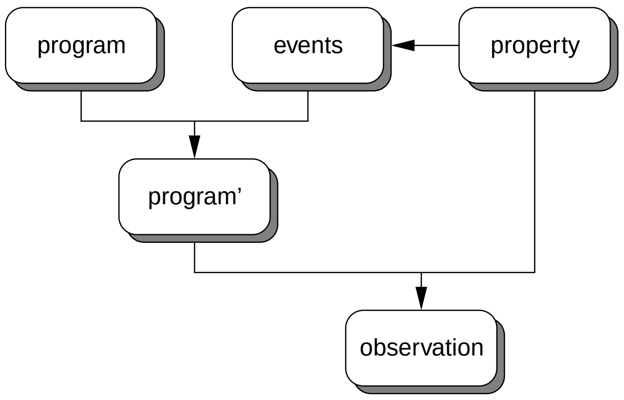

Figure 1.7: The runtime verification process

Validation

The process of runtime verification is illustrated in figure 1.7. The properties that are to be validated are stated formally be the person carrying out the test. Typically, these properties are inspired by the informal properties stated in the requirement specification of the control program. The properties must define high-level validation rules from low-level programming events.

In the MaC tool ([KKL$^+$01]), the specification language is divided into to parts: The *Primitive Event Definition Language* (PEDL) and the *Meta Event Definition Language* (MEDL). PEDL expressions define primitive events (such as a variable declaration, a variable assignment, or the entering of a method) and primitive conditions from these (such as event `a` holds until event `b`). In MEDL specifications, higher level events and conditions are created from those imported from a PEDL specification. Furthermore, the properties that must hold for the system are stated in terms of these events and conditions.

Next, the set of events that must be monitored has to be deduced. Often, the events that must be monitored can be deduced from the properties. In the MaC tool ([KKL$^+$01]), the event set is given by the primitive events defined in the PEDL script. Given the event set, small pieces of code emitting output describing the events are then inserted into the program to be tested. This modified (or *instrumented*) program is then executed and a monitoring unit observes whether or not it satisfies the properties defined.

Quality and Applicability

The error types supported by runtime verification range from runtime errors, such as program exceptions, to the validation of properties, typically specified in some temporal logic. Common properties include the existence or nonexistence of some event in a trace and that one event is followed by another at some time in the future. Noteworthy, some properties such as deadlocks and data-races in multi-threaded programs may be discovered without the need to explicitly specify a property. This is accomplished by observing the sequences of acquirings and releases of semaphores by different threads. If two threads acquire exclusive access to two shared resources without obeying to some ordering, there is a possibility of deadlock – even though it may not occur in the run.

The coverage of a run is measured in the percentage of code-lines executed and may vary significantly between runs. In general, the coverage experienced is low since validation is based only on a single run and the program is not *guided* to ensure coverage of abstract system state. As a consequence, false positives are common, and it can never be concluded that a program is correct with respect to some property on grounds of an error-free trace. The validation method is also prone to false negatives. The implicit checking of concurrency-related errors may identify, for example, a possibility of deadlock (without actually observing one) – but no current practices are able to prove whether or not they can in fact occur in a given program (unless actually observed).

The only manually performed task is stating properties to be checked, since altering the program representation can usually be performed automatically according to the property to check. This implies that runtime verification has a high degree of automation. Furthermore, since it is based on a single run of the program, it scales very well to large systems.

An issue of practical nature is related to runtime verification of real-time systems. The inserted code emitting events from a program consumes processing time – as well as does the observation process (including the maintenance of information about processes timing requirements). Of course, while these tasks are executed, time progresses, which may influence whether or not a process meets it timing requirements. A common solution to this problem is to ensure that the time spent for these activities are negligible with respect to the execution of the program under test.

Representation of the Environment

The tools for runtime validation that we have encountered all depend on the program being executed in its actual environment. That is, if a program responds to the press of some button, a user is required to physically push that button. Similarly, if a program generates some output to a physical device, the physical device must be present in order to receive the signal, thereby altering the state of the surrounding.

1.4.3 Model-Based Testing

Like runtime verification, *model-based testing* operates directly on the system by executing actual application code. However, as depicted in figure 1.8, instead of observing whether or not a given property is satisfied on a single run, in model-based testing a number of test, making up a *test suite*, are generated from a specification. The overall aim is then to test whether the implementation under test conforms to the given specification, that is, the implementation is correct with respect to that specification.

Validation

The specification is given as an *input/output model* – differing from the models previously discussed in that a transition may either emit an output to – or expect some input from – an external entity. The tests generated contain sequences of input to be given to an implementation, and a number of allowed outputs. In order to pass, the actual output, observed when

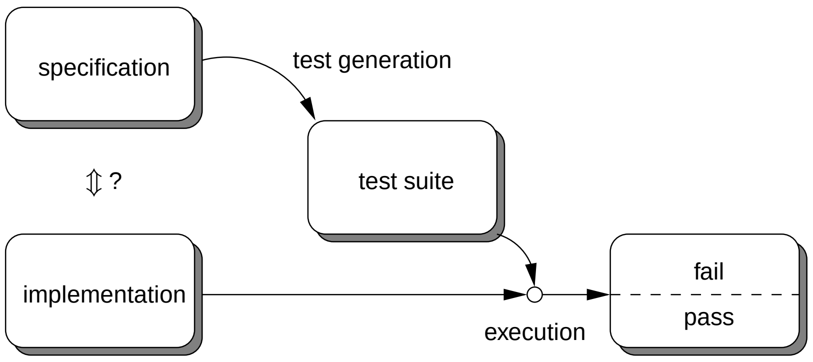

Figure 1.8: The process of model-based testing

executing a test, must be a subset of the allowed outputs. Furthermore, the test must emit no output *only* if no allowed output exists in the specification (referred to as *quiescence*).

If all tests pass, it may be concluded that the implementation conforms to the specification, under the following two conditions: First, the test execution must be *sound*, that is, no correct implementation will ever fail a test. Second, testing must be *exhaustive*, that is, every incorrect implementation will fail at least one test in the test suite. The latter objective is, of course, not obtainable in the general case.

Tests are generated from the specification model. In TorX ([dVT00]), tests are generated without input from the user. When in a given state, one of the following three options are chosen at random:

- *Offer input.* The implementation is given an input. This can always be done since models (and thus implementations) are expected to be *input-enabled* – that is, always accepts all input;
- *Expect output.* The observation unit awaits an output from the implementation;
- *End test.* End a test case.

Quality and Applicability

The coverage of model-based testing can be measured in the coverage of the model as well as in the percentage of the total amount of code lines executed. The coverage obtained relies heavily on the algorithm used for generating tests from specifications – a process whose description lies beyond the scope of this introduction. Though no measurements of coverage in current tools have been found, complete coverage is expected to equal exhaustiveness – and therefore be unobtainable. However, the coverage is expected to be better than that of runtime verification since each test in the test suite may be seen as a trace in runtime verification. As a consequence scalability is assumed to be worse than that of runtime verification.

Concerning the types of errors that can be detected, runtime errors as well as miscalculations by algorithms (in form of output) may be discovered. The remaining characteristics are comparable to that of model checking: A model must be manually constructed – but once this is accomplished it may be used for automatic testing of several different implementations. Also, the false negatives and positives experienced relies on the specification and the coverage obtained.

Tools for performing model-based testing (such as TGV ([JCTG96]) and TorX ([dVT00])) often perform the generation of test suites *on-the-fly*. That is, the tools interact with the implementations under test during execution, feeding them with input and observing the output produced. If the output is allowed, a new input may be given to system. Of course, such on-the-fly techniques have the same problems dealing with real-time requirements as do runtime verification.

Representation of the Environment

No explicit model specifying the behaviour of the environment is used for model-based testing. Instead, the algorithm for generating test cases uses the specification of the control program to decide the inputs and outputs to offer and expect. Due to the assumption of input-enablednes, the test case algorithm can offer any input to the implementation under test in any state.

By this approach, no assumptions about the behaviour of the environment is made. Thus, the control program can by subjected to all possible behaviours of an environment. It is assumed that the control program reacts only to certain inputs, namely those in the specification. If, during testing, it reacts to other inputs than those allowed by the specification, the test is concluded to have failed.

1.4.4 Limitations in Current Practices

Comparing the current practices described above, it is clear that a major difference lies in the trade-off between coverage and scalability. The high abstraction level and complete state space exploration of model checking allows for full coverage at the cost of scalability. At the other end, runtime verification operates only on a single run of the application, yielding good scalability and low coverage. Model-based testing has a better coverage than runtime verification, but lower coverage than model checking. Also, the coverage may be defined in both the amount of code executed and the part of the specification model explored. We identify the lack of good coverage combined with good scalability as a limitation in current approaches.

A second limitation we shall emphasise is the applicability of the validation approaches to real-time systems. While model checking tools exist for verifying properties of models real-time systems a large class of errors, namely those related to the actual execution of application code on a given platform, is not supported. For real-time systems this includes *performance* related errors – that is, errors where some timing requirement cannot be met on a given platform due to performance problems.

In runtime validation and model-based testing such performance related errors *can* be detected – but the time consumed observing whether properties are satisfied or not must be shown not to interfere with processes meeting their timing requirements. In conclusion, significant limitations exist in the applicability of current validation approaches to real-time systems.

Of the three current practices, only model checking incorporates an explicit specification of the environment. Current runtime based approaches assume that the program is executed in its actual environment. Since the behaviour of a physical environment is hard to control, assuring that the control program is subjected to a variety of different behaviours in the environment is difficult.

Model-based testing makes implicit assumptions about the behaviour of the environment based on the specification of the control program. This, however, does not allow impossible behaviours to be disregarded. Therefore, the control program is likely to be subjected to tests of environment behaviour that can never actually occur. Although it is important that the control program react only to the desired inputs, testing it for environment behaviour that is known never to occur constitutes an overhead in the testing process.

1.5 Proposed Validation Method

In this report, we propose a validation method that is primarily based on run-time validation. However, instead of interacting with a physical environment, the control program interacts with an *environment simulator* (see figure 1.9). This simulation is based on a model of the environment behaviour, comparable to the automata describing environment interaction used when model checking real-time systems. As earlier noted, embedded systems often operate in well-defined and well-described environments. Thus, obtaining a formal model of the environment behaviour can be accomplished by formalising the informal environment descriptions.

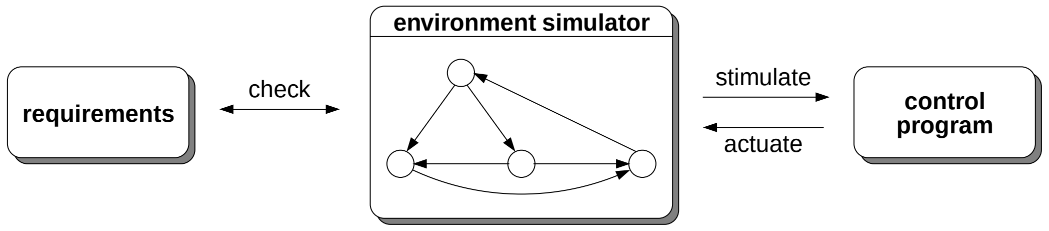

Figure 1.9: An illustration of the proposed validation method

In current run-time validation techniques, the requirements for the behaviour of a control program are given as properties in terms of the allowed program states. As illustrated in figure 1.9, we propose that the validation is based on the happenings of the environment – which it is the purpose of a program to control. By following this approach, the requirements can be stated as limitations in the allowed happenings of the environment, rather than invocation of methods and values of variables.

Specifically, by incorporating timing requirements into the specification of the environment, validation of control programs with real-time constraints is supported by the proposed method. By supporting a high-level abstraction of the environment and supporting validation of real-time systems in it through simulation, we aim at improving the applicability and quality of run-time validation for embedded systems. In the next sections, we investigate these properties for the proposed method.

1.5.1 Quality and Applicability Considerations

The proposed method combines some of the strengths of other practices. Most notably, the explicit modelling of environments known from model checking is applied to runtime verification. In this section, we investigate the quality and applicability of the proposed method, based on the measures defined in section 1.3.

Quality

The error types that can be found with the proposed method are equal to those supported by run-time validation. That is, they are in the categories of run-time errors, algorithmic errors, and timing errors. The fact that a model is used for representing the environment introduces a risk of false negatives, in the event that this model does not correctly describe the behaviour of the actual environment.

Coverage can be measured in terms of the environment model. Since the part of the behaviour of the environment that a control program is subjected to during validation is effected by the simulation algorithm, the coverage of the validation can in influenced by guiding the simulation. In chapter 4, we shall consider several algorithms for performing this guiding.

Applicability

The formalisation required in order to specify the behaviour of the environment must be performed manually. However, since an informal description of the environment is likely to be given, this task does not constitute a large manual effort. The same applies for the requirements of the environment. Given the automatic simulation and validation suggested by the proposed method, the degree of automation is relatively high.

The level of abstraction is equally high, since models of the environment and requirements in terms of these models are used for the validation. The scalability of the proposed method is assumed to be better than that of model-checking, since the exploration performed is not exhaustive. However, it is likely to be worse than the scalability of run-time validation since a higher coverage of the states is likely to be achieved.

1.6 Project Definition

In order to realise the validation method described in section 1.5, a number of tasks must be performed. In the following, we describe these tasks, and, in the next section we define the scope of the project. That is, we describe to which level of detail the tasks are covered in this report. These tasks are:

- *Environment Analysis.* In order for the environment to be described formally, an analysis of the generic patterns in behaviour of an environment must be performed. The various evolutions of environments over time must be captured in the formal models, in order to allow a simulation to be based upon them. The behaviour of an environment, of course, is dependent on its interaction with the control program. Thus, an investigation of the communication patterns between these entities must also be carried out;

- *Formal representation of the environment.* Having defined the behaviour that a formalism must be able to express, an appropriate formalism must be chosen. The formalism must support interaction with the control program and validation of environment requirements;

- *Simulation based on formal models.* The formal representations describe how environments may evolve over time under influence of the control program. Since the environment may change autonomously, a method for actively updating the state of the environment is necessary. Similarly, methods for interacting with the control program during execution are required;

- *Validation based on environment requirements.* When the control program is executed in parallel with the simulated environment, an approach for validation is required. This task consists of two subtasks: Obtaining a good coverage of the possible environment behaviour, and checking that a set of properties is satisfied. Both the coverage measure and property validation are based on formal models of environments.

1.6.1 Project Scope

In this project, we mainly focus on the analysis of environments, their formal descriptions, and the simulation based on these. We aim at obtaining a high level of automation, thereby minimising the effort required by test teams in order to use the approach. We also consider different approaches to assuring a good coverage of the validation, and how to allow validation to be performed on the states of the environment.

Since the area of expressing properties for runtime validation is well-researched in the MaC tool ([KKL$^+$01]), we choose not to go into depth with property expression (see section 1.4.2). However, relatively simple expressions for restricting the allowed states of the environment will be supported.

Furthermore, we design and implement a prototype, in which the functionality requirements deduced via the analysis is supported. Based on this prototype, a case study of a control program for the operation of bridge is performed, in order to identify strengths and weaknesses of the proposed validation method.

1.6.2 Outline of the Report

The work presented in this report is a continuation of the work presented in [Chr02]. A significant part of this introduction is a modification of the introduction presented in the previous report. Chapter 2, an investigation of the characteristics of real-time systems, was published in an almost identical version in the previous report. The remainder of the report is structured as follows:

- *Chapter 2*, ***Real-Time Systems***, gives a description of the characteristics of real-time systems, including a definition of the concept, timing scopes, and an overview of scheduling policies. Furthermore, a classification of the error types that can occur in real-time systems is made.
- *Chapter 3*, ***The Inquisitor Framework***, describes the functionality provided by the framework resulting from the work presented in [Chr02]. The framework allows runtime validation of real-time systems implemented in conformance with the Real-Time Specification for Java.
- *Chapter 4*, ***Analysis***, includes the analysis of general environment behaviour and interaction. A formalism for modelling the behaviour of environments is chosen, and a method for simulating these models is presented. This method allows validation to be performed in terms of the states of the model.
- *Chapter 5*, ***Design***, describes how the general simulation and validation approaches described in the analysis can be implemented in the Inquisitor Framework.
- *Chapter 6*, ***Case Study***, contains the case study of the control program for performing the operations of a bridge. Based on this implementation, a test of the prototype is performed.
- *Chapter 7*, ***Conclusion***, contains a conclusion on the work presented in this report and describes future work.

Real-Time Systems — Chapter 2

In this chapter, we define and characterise real-time systems in order to obtain a well-defined semantics of terms used in the remainder of the report. The chapter also serves as a theoretical background for the description of the Inquisitor Framework, given in chapter 3.

After defining real-time systems, we introduce an example real-time system that sorts bricks on a conveyor belt, which will be used throughout the next three chapters. In accordance with this, many of the examples of the characteristics of real-time systems in this chapter will be given in reference to the brick sorter example.

Having described the characteristics of real-time systems we proceed to investigate the types of errors that real-time systems may exhibit. This categorisation is made with the second objective of the report – validation of Real-Time Java applications (as defined in chapter 1) – in mind. We consider it useful to differentiate errors by category when validation strategies are evaluated.

2.1 Definition

In chapter 1 we introduced real-time systems as systems in which *the overall correctness depends not only on the correctness of results produced but also on the correct timing of these results*. This definition, however, does not explicitly state anything about the environment in which such systems reside. For our purpose of validation we wish to distinguish real-time systems from their environments. We therefore adhere to a modification of the following definition by the Predictably Dependable Computer Systems project:

> A **real-time system** is a system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment in its current state.

Figure 2.1 depicts this definition of a real-time system. Given a stimuli from the environment at time t a real-time system must react within an interval $[\Delta;\Delta+\Delta']$, $\Delta,\Delta' \in \mathbb{R}^n$ after t. The modification of the definition is that we explicate that the current state of the environment is allowed to effect the appropriate reaction as well as the interval of time (that is, the values of Δ and Δ') that this reaction must occur within. Since the current state of an environment decides the possible future states, changes in the environment may also influence what is considered correct behaviour. Note that, by the definition, stimuli may simply be that an amount of time t_1 has passed. Thus no explicit *input data* is required from the environment in order to constitute stimuli.

Real-time systems are often categorised with respect to the importance of reacting within the proper interval of time after a given stimuli. Systems where such misses are fatal to the continuance of the system (because of damage to the environment) are said to be **hard** real-time systems. An example of a hard real-time system is a control program monitoring some process (e.g. chemical or nuclear), terminating the process if it starts to run out of control. Failure to react to such runaway processes may have disastrous results.

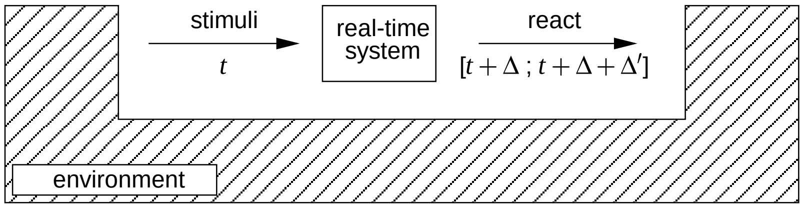

Figure 2.1: A real-time system interacting with its environment

If, on the other hand, a system is able to survive such misses, the system is characterised as **soft**. Examples of soft real-time systems include multimedia systems, where a temporary failure to play back video or audio files with the correct timing may result in bad quality – but does not prevent the system from continuing the execution.

2.2 An Example – The Brick Sorter

In this section we introduce an example that contains properties characteristic to real-time systems (described in the next section). The example, depicted in figure 2.2, consist of a conveyor belt, a (light) sensor, a piston, and red and black bricks. These entities make up the environment of the system. The conveyor belt moves bricks at a constant speed from the entry point at the left towards the exit point at the right.

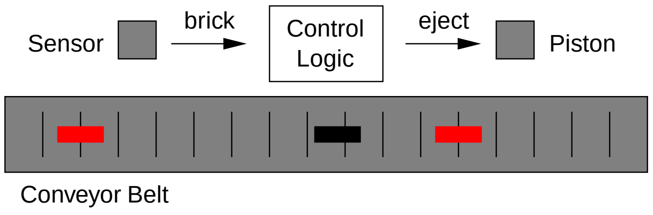

Figure 2.2: The brick sorter

The aim of the system is to eject all red bricks from the conveyor when passing the piston. Black bricks should be allowed to reach the exit point. We assume that the highest frequency with which bricks can be placed on the conveyor belt is known – and that this frequency is low enough that no brick will ever be placed on top of another. We shall further assume that the belt is either *running* or *stopped*.

The control program consists of some application logic controlling when to eject bricks. The control program may poll the sensor for the colour of the brick in front of it. If no brick is located in front of the sensor a sensor poll will return a special value expressing this fact. The piston supports an eject action.

If a brick is located in front of the piston at the time of an eject action, the brick will be pushed off the conveyor. No information about the position of a brick can be obtained between the sensor and the piston. Consequently, when a red brick is observed in front of the sensor (stimuli), the time at which an eject should occur (reaction) must be calculated from the distance to the piston and the speed of the conveyor belt.

2.3 Characteristics

In order for a real-time system to react timely to given stimuli, it is necessary to be able to express (and schedule accordingly to) these timing demands in real-time software. Consequently, applications must be able to access clocks and express time. Furthermore, we introduce *temporal scopes*, by which we are able to express the desired specifications of timing demands. Finally, we describe the concept of periodicity – that is, performing a task several times with a given period between executions. The section is based on chapters 12 and 13 in Burns and Wellings ([BW01]).

2.3.1 Clocks

Clocks represent the *passage of physical time* in the environment, mentioned in the definition of real-time systems. They are discrete representations of continuous time. That is, time proceeds in *ticks* constituting the fact that a fixed period of time has elapsed. The amount of time that a clock is increased per tick is called the *resolution*. Typical resolutions are measured in nanoseconds or milliseconds. A clock may be queried by applications for its *current time* – that is the current value of the clock in some known unit of time.

Though multiple clocks may be available to applications, we shall assume the presence of a *global* clock. All other clocks must be in synchronisation with the global clock. That is, even though a given clock has a resolution different from that of the global clock, it still proceeds at the same speed. Without this assumption, we would have to take the phenomenon of *clock skew* (one clock drifting from another) into consideration when implementing the framework.

2.3.2 Expressing Time

Having access to a clock enables a real-time application to query the current time. Most applications, however, will also need the ability to express time in the past and future. For example, in a personal calendar application, the user may enter an appointment at a certain time of day. Furthermore, the application may provide functionality to pop up alerts on the screen some interval of time before appointments. We wish to be able to express such timings in real-time software. To this end we introduce the concepts of *absolute*, *relative*, and *rational* time.

An **absolute** time t with respect to some clock c refers to the time at which c's current time is t. Figure 2.3 (a) illustrates this. Note that the current time value of a clock is itself an absolute time. As an example, consider the application logic for polling the sensor value in the brick sorting example. If bricks were known only to arrive at certain pre-scheduled times, an absolute time could be used to express the next time the sensor should be polled.

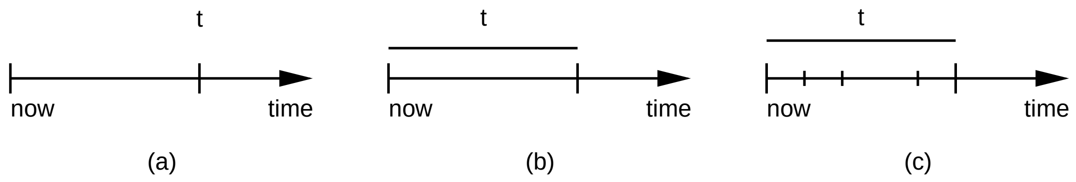

Figure 2.3: (a) Absolute time, (b) relative time, and (c) rational time

A **relative** time t refers to a point in time relative to some absolute time offset (often the current time of the clock, implicitly given by the offset). Thus a relative time t with an offset at absolute time t' is equivalent to the absolute time $t' + t$. Note that relative times may be negative as well as positive values. In the brick sorter example, the time the eject signal should be sent to the piston can be found relative to the time a red brick is sensed, and thus, a relative time may be used to express the time of ejection.

Rational time can be expressed as a pair (t, n) where t is a relative time and n is positive integer. The semantics is that *n events* should be distributed over the interval of time given by t. The distribution of these events is not necessarily even. In the illustration of rational time $(t, 3)$ on figure 2.3 (c) the events are clearly not evenly distributed.

2.3.3 Temporal Scope of a Process

Recall the requirement from the definition of real-time systems to *react to stimuli [...] within time intervals dictated by the environment*. Now that we have described how applications may express time, we introduce terminology that allows us to describe the timing requirements of a process. The definition of the timing of a process is called the *temporal scope* of that process. Introducing temporal scopes of processes shall allow us to refine our categorisation of real-time systems.

In the following we shall understand by a *process* an active unit executing a task. The temporal scope of a process can be illustrated as in figure 2.4. The figure is inspired by [BW01] fig. 12. The following terms are used to describe the timing of a process' execution:

- *Now* – the absolute time from which the temporal scope is defined;
- *Minimum Delay* – the earliest time relative from now at which a process may be released;
- *Maximum Delay* – the latest time relative to now at which a process may be released;

- *Release* – the actual time a process starts executing its task;
- *Maximum Elapsed Time* – the maximum amount of time that is allowed to pass before the process must finish its task after release;
- *Worst Case Execution Time* – the guaranteed longest time a process will consume executing its task. The value is often required to be supplied by the developers. Depending on the scheduling policy used (See section 2.4), a process may or may not be *preempted* during execution. This is illustrated in figure 2.4 by the disjoint intervals of time t_1, t_2, and t_3 (during which the task is executed). The sum of these equal the worst case execution time;
- *Deadline* – The latest time relative from now a process is allowed to finish executing its task. In order to be correct, a process must finish executing its task before the deadline – regardless of the amount of time the process has been in a preempted state.

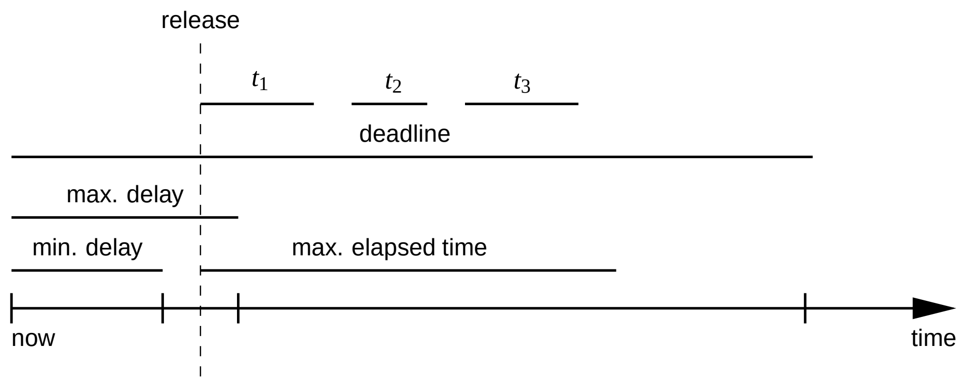

Figure 2.4: The temporal scope of a process

In accordance with the categorisation of hard and soft real-time systems processes are categorised as either *soft*, *firm*, or *hard*. We shall call a process **soft** if the following hold: One, Failing to deliver a result between the end of the delay and the deadline inflicts no harm on the system and its environment, and two, the system will benefit from the result despite its untimely occurrence. **Firm** processes differ from soft ones only in the second clause. In firm processes no benefit is drawn from untimely results. **Hard** processes are those where failing to deliver results within the window specified by delay and deadline will prevent the system from continuing execution.

The categorisation of processes allows us to refine our categorisation of real-time systems. We shall call a real-time system *soft*, *firm*, or *hard* depending on the category of the *hardest* process in that system. Thus, if the hardest process of a system is firm we categorise the system itself as firm.

A common way of delaying the execution of a task is by the use of *timers*. A **timer** is specified by a *fire time* at which one or more *associated tasks* are processed. Returning to the brick sorting example we notice that we may use a timer to eject bricks. The timer is started when a red brick is sensed and fires an interval of time (equal to the time it takes the brick to move from the sensor to the piston) later. When the timer fires, a task that sends the eject signal to the piston is processed.

2.3.4 Periodic and Aperiodic Processes

Often a task must be performed repeatedly by a process. If a process performs its task at a fixed interval of time we shall call the process **periodic**. When a process reaches the end of a period (given by an amount of time relative to now) the value of *now* is increased by an amount of time equal to the period. After an appropriate delay, the task is then re-executed. Since we view a process as a single active unit, the task must be finished before a new period can start. Thus, in the temporal scope of a periodic process, the deadline cannot exceed the period. If we need a process to finish later than the end of a period, the periodic process may spawn a new process to handle the task.

Processes that are not periodic are called **aperiodic**. The fact that a process is not periodic does not prevent it from performing its task multiple times. For example, the task may be executed at the occurrence of an event given by a rational time. We shall call timers periodic if they fire multiple times at fixed intervals and aperiodic otherwise.

2.4 Scheduling Real-Time Systems

In the previous section, the *temporal scope* of a process was introduced with the purpose of describing the timing requirements of processes. The temporal scope included the time of *release* and the possibility of a process being *preempted* – both of which were external to the process in the sense that the process does not have control over the occurrence of these events. Performing these external tasks is the responsibility of the *scheduler*. As a consequence, the *scheduling policy* employed is important to the overall correctness of a real-time system.

Before describing some commonly used scheduling policies we shall make some assumptions about the scheduling model and elaborate on the preemption of processes. Furthermore, we shall introduce the concept of *schedulability*.

2.4.1 Scheduling Model

We shall assume that only a single unit of execution is available. That is, all processes are executed on the same processor and only one process can be executed at a time. With this assumption it is clear that no *parallelism* (two processes executing at the same time) can occur. However, processes may still execute *concurrently* since processes can be preempted during execution.

When a process is **preempted** by a scheduler it is removed from the processing unit and cannot proceed until reinserted by the scheduler. Such points of preemption are under the complete control of the scheduler. Often, real-time programming languages have facilities that enable a process to inform the scheduler that another process should become running.

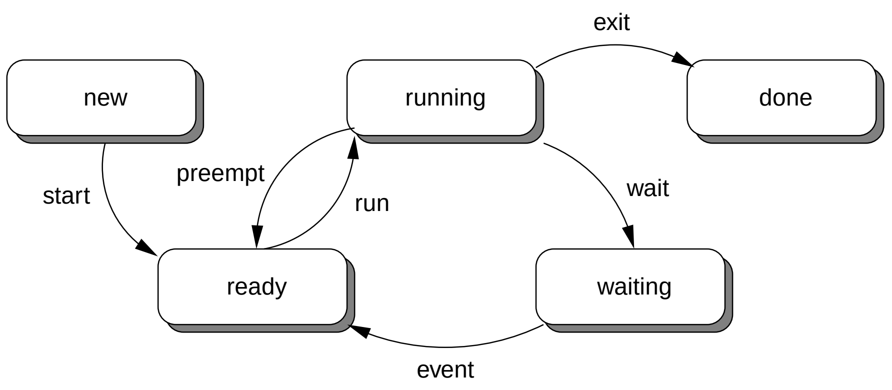

Figure 2.5: Process state based scheduling model

Typical examples are functionality for letting another process (chosen by the scheduler) become running and waiting for access to a resource or some event to occur.

In compliance with the processing model of Real-Time Java, we assume that schedulers are *process state based*. This model is illustrated in figure 2.5. A process can be in one and only one state at a time. In the following, the states of the model and the transitions between them is explained:

- *New* – When a new process is created its initial state is new. The process becomes ready for execution when the creating process starts it.
- *Ready* – Ready processes are those that are ready to be executed on the processor. In the literature, ready processes are often referred to as *runnable*. Due to the naming clash with the Java interface `Runnable` we have chosen to use the term ready.
- *Running* – When a process is executed on the processor we refer to it as running. Since we assume a single unit of execution, only one process can be running at a time. A process becomes running when the scheduler selects it from the set of ready processes. If, at some point, the scheduler chooses to preempt the running process (or the process *yields* to other processes), the process will become ready again and another process is selected for execution.
- *Waiting* – Running processes may need to wait for some event to occur. Such events include awaiting the firing of a timer or access to some resource. For now, we shall not distinguish such waits, but the Inquisitor Framework does. When the event occurs the process becomes ready for processing.
- *Done* – When a process finishes its work we call it done.

An important concept is that of **schedulability**. A set of processes are said to be *schedulable* if there exist a schedule that allows for all processes to execute correctly with regards to their timing scopes. Depending on the scheduling policy used, there exist different algorithms for testing schedulability statically. These tests often rely on the worst case execution times provides by the developer. We shall not be going into any depth with schedulability algorithms, since the purpose of the framework includes testing real-time applications for schedulability dynamically.

2.4.2 Overview of Scheduling Policies

The scheduler controls the states of processes, as illustrated in figure 2.5. We refer to the various algorithms for such control as **scheduling policies**. The choice of scheduling policy is a trade-off between abstraction and control. The lower the level of abstraction, the more control the developer has over the scheduling. In this section, we shall describe three different scheduling policies and their characteristics. We then select *priority-based preemptive scheduling* as our policy of choice, and describe it further in the next section.

A simple approach to scheduling is that of *cyclic* dispatching. The approach, also known as **round-robin** scheduling, selects processes in a cyclic manner, allowing them an equal amount of time to execute. Thereby, the policy incorporates *fairness*, that is, no process is *starved*, in the sense, that it is never allowed to run.

Whereas round-robin scheduling is common for applications without real-time requirements, it often comes in short for real-time systems as exemplified in the following. Consider a real-time system containing two processes, p_1 and p_2, with deadlines, t_1 and t_2, respectively. Further assume that $t_2 < t_1$ and that the worst case execution time of p_1 exceeds t_2 relative to the current time of the global clock. Now, if both p_1 and p_2 are both ready, p_1 is as eligible for execution as p_2 – even though dispatching p_1 may prevent p_2 from finishing its task before the deadline.

From the example above, it is clear that in order to obtain a more reasonable scheduling, processes should be assigned *importance* – allowing the most important process to execute before others. One way to implement this is to order processes by their deadlines – making the process with the earliest deadline the most important. This policy is called **earliest deadline first** – or *EDF* – scheduling.

EDF scheduling gives the developer greater control over the execution of processes in a system (due to the increased predictability), without requiring explicit assignment of importance to processes. For instance, the problem mentioned in the example used to illustrate the shortcomings of round-robin scheduling is no longer an issue when EDF scheduling is employed.

There are, however, limitations to this implicit ordering of process importance. In a hard real-time system, a soft process may be executed prior to a hard process even though this may prevent the hard process from meeting its deadline. In such cases, the programmer must declare the importance of processes explicitly. Such scheduling policies are known as **value-based scheduling**.

We investigate a kind of value-based scheduling where each process is assigned a **priority**. This priority represents the importance of the process – higher priorities correspond to greater importance. Priority-based schedulers can be categorised with respect to the allowed priority assignments for a given process:

- *Static* – Fixed priorities are assigned to processes statically, such that, each time a process is released, it has the same priority;
- *Dynamic* – The priority of a process is determined at the time of its release. It remains fixed throughout the execution of its task;
- *Adaptive* – Processes may change priorities during execution of tasks. The reassignment may be performed by the process itself, or other processes (during preemption of the process who's priority is being changed).

Static priority-based scheduling is suitable for well-defined environments, whereas dynamic and adaptive priority-based scheduling are often necessary in cases with greater dynamics in the environment. For example, the priority of a process may depend only on the state of the environment at the time it is created (and explicitly not of the future state), in which case dynamic scheduling is applicable. If the task performed by a process can become more or less important due to changes in the environment during its execution adaptive priority-based scheduling is required.

In the Inquisitor Framework, we chose to use a priority-based preemptive scheduling policy that allows adaptivity of priorities. This choice is based on the following observations:

- It is the default scheduler required by the Real-Time Java Specification;
- It provides greater control over processes than do EDF and round-robin scheduling. This is favourable since it adds to the generality of the framework;
- It is the most expressive priority-based scheduling policy. That is, the developer of an application may choose to use only static or dynamic assignments of priorities, if it suffices for the needs of that application.

For the remainder of this report we shall always assume adaptivity when referring to priority-based preemptive scheduling. The following section describes the scheduling policy further and introduces the problem of priority inversion.

2.4.3 Priority-based Preemptive Scheduling

The guiding principle of priority based preemptive scheduling is that the process in the ready set with the highest priority should always be running. If several processes with priority equal to the highest exist one of these should run. If a process becomes ready due to the end of a delay or access to some resource and its priority level is higher than that of the running process a process switch should occur (by preemption of the running process).

When processes with different priority levels share resources, a situation where a high priority process is forced to wait for a resource held by a lower priority process can occur. A scenario where this happens is depicted in figure 2.6. Let process a have a lower priority than process b – which in turn has a priority lower than process c. Say that process a is released first (due to its temporal scope) and obtains exclusive access to some resource r.

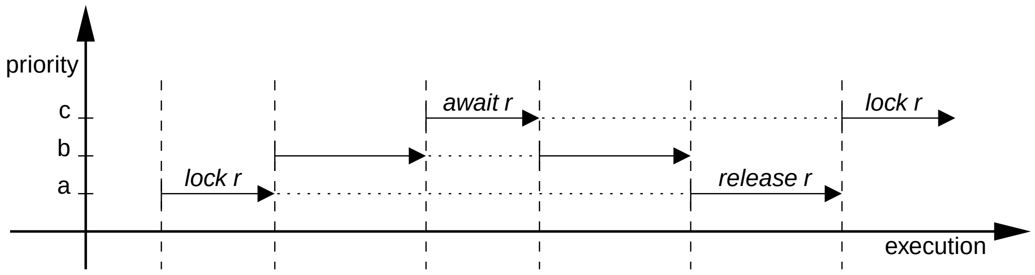

Figure 2.6: An illustration of priority inversion

After a while, process *b* is released and *a* is preempted due to its lower priority. Process *b* now runs until *c* is released and requests access to *r*.

Now, an unfortunate situation exists where *c* cannot get access to the shared resource *r* as it is held by *a*, and *b* is released again since it is the ready process with the highest process. Such behaviour is undesirable since *c* has the highest priority of all processes and could become running if *a* was allowed to finish its task and release *r*.

The above phenomenon is known as **priority inversion** since *c* is forced to wait for lower priority processes to finish. Cornhill *et al.* introduced an algorithm for avoiding priority inversion using *priority inheritance* ([CSL$^+$87] pp. 33-39). Basically, processes holding resources required by higher priority processes *inherit* the priority of the higher priority process. Thus, in the example of figure 2.6, after *c* requests access to *r* the priority level of *a* will be changed to that of *c* – allowing *a* to proceed ahead of *b*.

2.5 Error Categorisation

For our purpose of validation, we regard it as important to classify the types of errors that real-time systems can exhibit. First of all, it is important for the cause of establishing which classes of errors can be detected using different validation methods. Second, knowledge about which errors may exist is important for the choice of validation strategy. Finally, the classification may prove useful when stating properties that should hold for a given system and generating relevant output for debugging if an error is found.

We divide errors into four classes. The first two classes, *application logic* and *concurrency-related* errors, are known to exist in systems without real-time requirements. The latter two classes classify timing errors with respect to their dependency on a given platform. The error classes are described in the following:

- *Application logic errors* – Errors that may be described as *functional* errors in the application logic. That is, given some input, the program responds in an erroneous way. Such errors include producing incorrect outputs and entering wrong (possibly fatal) states, resulting in incorrect termination of the application;

- *Concurrency-related errors* – Well-known errors that may arise when applications use concurrent processes. This includes *deadlock* when processes acquire locks on resources without adhering to some ordering of the resources. Lack of synchronisation on resources may lead to *race conditions*, and the presence of greedy processes may lead to *starvation*;

  Whether a deadlock or race conditions occurs or not depends on the relative timing of processes. Thus, even though processes p_1 and p_2 lock resources r_1 and r_2 without proper ordering, deadlock may not arise if, for example, p_1 and p_2 cannot logically execute concurrently. Knowledge about the relative timing of processes may be accessible via the temporal scopes of processes. In conclusion, if p_1 and p_2 can be shown never to execute concurrently, we may deduce that deadlock cannot occur and thereby eliminate some of the false negatives mentioned in section 1.4.2;

- *Timing-related logical errors* – Platform independent errors where reactions to stimuli occur with incorrect timing. An example of this is when the timing scope of a process is incorrectly defined. A second example would be one process suppressing the execution of another, more important process, due to a higher priority (possibly due to a lack of adaptivity to changes in the environment).

- *Performance-related errors* – Even though the timing logic of a real-time application is correct, errors related to timing may still occur on a given platform. For example, if the worst case execution time of a task does not hold on a given platform an error may occur. We shall refer to such errors as *performance-related*.

The Inquisitor Framework CHAPTER 3

In this chapter, we describe the functionality provided by the Inquisitor Framework, presented in [Chr02]. The framework is a partial implementation of the Real-Time Specification for Java. First, we give an introduction to the functionalities of the specification that have been implemented. In addition to scheduling Real-Time Java programs, the framework includes supports for performing validation of real-time systems in simulated environments. We present these features and give an overview of how the validation is performed by an example.

3.1 Real-Time Features

The Real-Time Specification for Java ([RTJ02]) includes a great variety of initiatives for supporting the implementation of real-time systems in Java. The Inquisitor Framework contains support for the scheduling of real-time systems implemented in accordance with this specification. However, only a subset of the features in the specification has been implemented:

- *Preemptive, priority-based scheduling policy.* The framework is based on a preemptive, priority-based scheduling policy. The scheduler performs the task of scheduling so-called *schedulable* entities. These entities have objects representing their priorities and timing scopes associated with them. A major difference between the scheduling of Real-Time Java programs compared to regular Java programs is that the thread with the highest priority must also be executing if possible. Similarly, when using Java's built-in support for thread synchronisation, the thread of highest priority must be the most eligible for the semaphore;
- *Definition of timing scopes.* The timing scopes associated with schedulables entities which allows them to define aperiodic and periodic timing scopes;
- *Access to time through clocks.* Programs have access to clocks from which time can be represented in absolute and relative terms. Among other uses, the classes representing absolute and relative time are used for the definition of timing scopes as well as timers;
- *Timers.* The specification allows special timer classes, which trigger executions (possibly periodically) of so-called *event handlers*. Such handler are represented by the class `AsyncEventHandler` and include a method which performs the given task.
- *External happenings.* Similar to asynchronous events being triggered by timers, event handler may also be bound to a so-called *external happening*. These happenings are handled directly by the virtual machine and result in the triggering of an event handler.

The functionality described above has been implemented in Java. That is, the added features for expressing real-time characteristics are implemented on top of an existing virtual machine. Some of the functionality normally placed in the virtual machine has been re-implemented in Java code to allow the new semantics imposed by the specification.

The two most prominent examples of this re-implementation is the support for preemption and the increased predictability of thread synchronisation. In order to achieve these features, it is necessary to explicitly invoke the methods supporting them. This, however, can be automatically performed by a process know as *instrumentation*. In this process, the source code of a program is investigated at the byte level, and byte code is inserted when appropriate in order to invoke the required logic. Though the instrumentation process is possible to implement in the framework, this has not yet been accomplished.

3.2 Validation Support

As mentioned, the Inquisitor Framework includes support for run-time validation of real-time systems. By this approach (depicted in figure 3.1), control programs interact with simulated environments as opposed to their physical surroundings. The control program consists of a number of processes interacting through a common interface with a set of processes simulating the behaviour of the environment. In this way, the control program is stimulated by another program, which also acts on output from the control program.

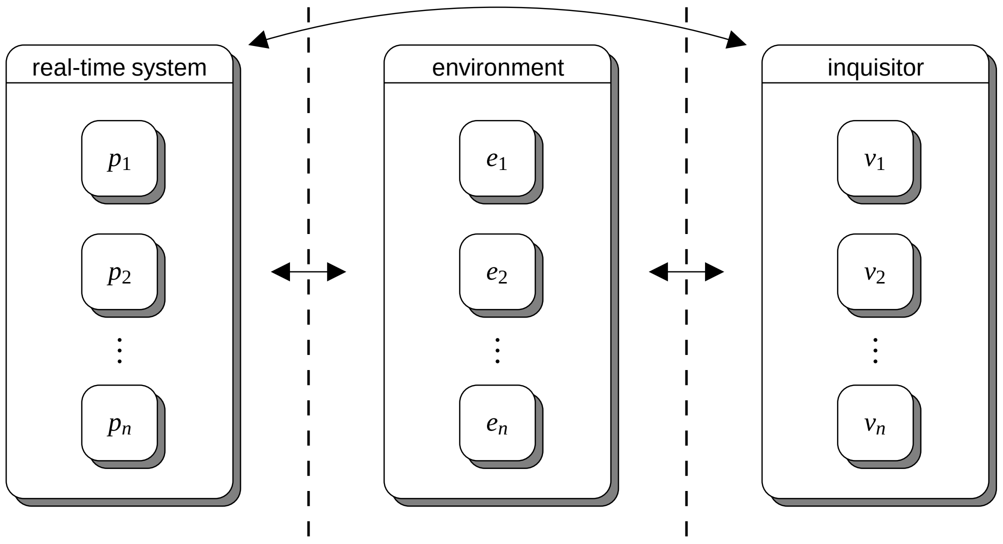

Figure 3.1: Validating real-time systems in simulated environments

In order to perform validation in the simulated environment, a third component, the *inquisitor*, is executed in parallel with the environment and control program. The inquisitor stimulates the program in accordance with the possibilities defined by the environment. Furthermore, it checks that the control program acts in accordance with some specification – for example a set of properties.

The inquisitor tasks are performed by special threads, referred to as *inquisitors*. These threads have all the capabilities of the schedulable entities of the scheduler. However, they have a higher priority than all other entities, which allows them to perform their task whenever it is required. In practice, to be an inquisitor, a Java process must extend the `Inquisitor` class in order to have the special characteristics.

By the validation approach described above, the control program, its simulated environment, and the inquisition component are executed in parallel on a single processing unit. This raises an issue regarding the timing scopes of the control program. When the environment and the inquisition components perform the updates, stimulation and checks, processing time is consumed. Thus, the timing scopes of the control program suffer – leading to possible deadline breaches which would not have occurred otherwise.

In order to handle this problem, the concept of *logical time* is introduced. As opposed to physical time, which progresses continuously, the progress of logical time may be stopped. By this approach, logical time is stopped whenever an inquisitor thread becomes running – and is restarted only when a control program thread again becomes running. In other words, logical time progresses if and only if a an inquisitor thread is not running. Logical time is implemented by manipulating the clocks used by Real-Time Java processes to gain knowledge about time.

3.3 Example

In order to illustrate how validation is performed in the inquisitor framework, we present an example how it is performed. The environment implemented is the brick sorting system described in section 2.2. The overall purpose is that red bricks are ejected and black bricks continue to the end of the conveyor belt. The application must therefore detect red bricks – at which point a timer must be started. When this timer fires, the piston should be given a signal to eject bricks in front of it.

We assume that a sensor must be polled in order to detect the colour of the brick in front of it (red, black, or none). The piston is triggered by sending it a an electrical signal. The total system is implemented as Java classes, some of which perform control program tasks, while others perform validation tasks. The system consists of the following parts:

- *Environment* – Three classes will represent the environment: `ConveyorBelt`, `Sensor`, and `Piston`. The `ConveyorBelt` class will be responsible for holding the current set of bricks (instances of a class `Brick`, with an attribute describing the colour) along with their current positions on the belt. At a given period, the positions of bricks are increased, simulating that the belt is continuously moving. In order to accommodate this, the `ConveyorBelt` class is made active – which means it must specialise `RealtimeThread` in order to be scheduled. However, since the thread simulates environment activity, it must instead specialise the `Inquisitor` class – which in turn extends `RealtimeThread`;

  The `Sensor` class includes a method, by which the control logic can poll the current sensor value, that is, the colour of the brick currently placed in front of it. One of three values – red, black, and none – is returned. The `Piston` class includes a method for ejecting bricks – which will remove the brick in front of it if one exist. The `Sensor` as well as the `Piston` class operate on grounds of the data represented by `ConveyorBelt`. Since both are activated by a thread in the control logic, instances need not be active objects themselves;

- *Control logic* – The control logic is represented by two classes. `PollThread`, which is a specialisation of `RealtimeThread`, performs periodic polling of the sensor by invoking the appropriate method on the `Sensor` class. The other class, `EjectHandler`, is a specialisation of `AsyncEventHandler` and is associated with a periodic timer, firing when a brick must be ejected, at the time of construction.
- *Inquisition unit* – The inquisition task in this example consist of a class specialised for validation of the control logic described above. The class, `BrickFeeder`, extends `Inquisitor` and performs the task of feeding the system input in form of instances of the `Brick` class. It also observes whether or not the system is well-behaved. In this case, this means that all red bricks are ejected and that no black bricks are ejected.

The system is validated by executing the control program, environment and inquisitor components in parallel. Whenever the control program needs to interact with its surroundings, it instead interacts with the simulated environment. When the environment must be updated or the inquisitor investigates the system, logical time stops to progress. Thereby, the timing scopes of the control program processes are not interrupted.

The above example is previously published in [Chr02] as part of the experimental work. In this work, erroneous mutations of the control program were tested in the simulated environment. The results of these tests can be found in chapter 5 of [Chr02].

Analysis — Chapter 4

Through our investigation of real-time systems in chapter 2 we found that real-time systems rely on interaction with their environments. In this chapter, we turn our attention to an analysis of the behaviour of such environments. The purpose is to find a method, by which the environment of real-time systems can be simulated from a formal representation of the environment. Furthermore, we describe different approaches to automatically exploring a given environment in order to obtain a good coverage of its possible behaviours. For fulfilling our aim of validation in simulated environments, we describe how requirements can be specified in terms of environment properties.

4.1 Behaviour and Interaction

Before considering which formalisms to use for modelling environments, we examine the generic behaviour and interaction patterns of environments. Based on this description, we define the environment characteristics that we require the chosen formalism to support through modelling, and give an example of such an environment. This example is used as a common example throughout the next two chapters.

4.1.1 Environment Behaviour

Consider the brick sorting example presented in section 2.2. The state of the environment changes over time in various ways. During normal operation, the belt moves forward at a uniform rate, thereby moving each of the bricks. However, if an engine malfunctions, the movement of the belt may suddenly stop. The event that an error occurs in an engine has a sporadic nature, contrary to the continuous movement of the belt.

We characterise these continuous and sporadic changes to an environment in the following way:

- *Continuous changes* perform evolutionary updates of the environment. The environment changes by some rate per time unit. This rate may itself change over time.
- *Sporadic changes* occur instantaneously without progression of time. Thus, no immediate change in the environment can be observed. However, the future course of the environment is effected. In other words, a sporadic change alters the continuous changes in the environment.

Some of the sporadic events of an environment involve the creation or destruction of entities in the environment. For example, in the brick sorting case, bricks are added to the system at the beginning of the conveyor belt. These bricks are *new* in the sense that they were not considered part of the system prior to their insertion. Similarly, bricks stop being part of the system if ejected from the belt by the piston, or, if they reach the end of the belt. Such dynamic creation and destruction, is, however not considered in the scope of this project.

4.1.2 Control Program Interaction

Figure 1.2 in section 1.1 illustrates how a control program interacts with its environment through sensors and actuators. The control program interacts with these devices in a data-oriented manner, allowing two kinds of communication:

- *Event-based* communication, by which a unique *signal* is used to inform the peer that some event has occurred. We shall refer to this type of communication as *emitting events* or *signalling*.
- *Value-based* communication, where values are sent and received. Typically, such communication is used when *polling* a sensor for the current value of some entity (for example, a temperature or position).

Although a control program sometimes uses value-based communication to actuate on its environment, we restrict value-passed communication to polling. Allowing control programs to send data to the environment would require the specification of the environment to contain procedures for dealing with the data received. Though this would be possible, we wish to simplify the model specification by not allowing it.

In summary, our requirements for communication of simulated environments are event-based communication in both directions and value-based communication in the form of polling.

4.1.3 Example

Since we do not allow dynamic creation in the environment, we now introduce an example that does not include it. The environment, illustrated in figure 4.1, will be used throughout this chapter as well as the next. It consists of a barrel with a water intake and an electrical pump connected to a water output. When the pump is off, the water level of the system increases over time, whereas, when the pump is on, the water level decreases.

We assume the pump to be cooled by the water it is pumping. If the flow of water stops during operation, the pump overheats and is destroyed. During pump operation, the water output may stop for two reasons. First, the barrel may run empty, and second, the water flow may be stopped by dirt in the pipes. In the latter case, the control program is informed by an event, `EV_ERR`. Once the error has been corrected, the `EV_OK` event is emitted.

In order to control the water level of the barrel, a control program observes the water level by polling the value of sensoring unit. The water level must be kept within an interval assuring that the pump is not running without a flow of water, while, at the same time, the water level does not rise beyond the capacity of the barrel, leading to an overflow.

The control program starts and stops the pump by sending it the `EV_START` and `EV_STOP` events, respectively. If, during pump operation, the `EV_ERR` event is emitted by the environment, the control program must stop the pump within a short time-limit to prevent it from overheating. The pump may then not be started again until receipt of the `EV_OK` signal.

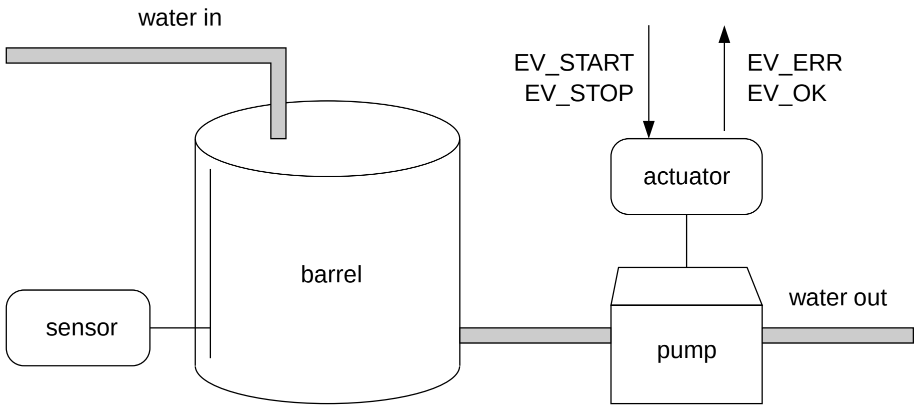

Figure 4.1: The barrel example

4.2 Formal Specification of Environments

Having analysed the behaviour and interaction of environments, we turn our attention to finding a method for specifying them formally. In this section we first state our requirements to the formal specification, after which we consider two candidates: Timed Automata and Hybrid Automata. Based on a comparison of the two, we choose to model environments as Hybrid Automata, although in a version modified for our needs. The remainder of the chapter is devoted to explaining how validation in a simulated environment can be obtained.

We require the formalism to be able to express sporadic and continuous changes alike. Since continuous changes happen over time, the formalism must also have some means for expressing the progress of time. We further require the formal models to allow interaction with a control program during simulation. This means that actuation and stimulation events must be explicitly modelled and that values that are to be polled by control programs must be available.

4.2.1 Formalisms

In this section, we consider two formalism that are both able to express real-time constraints. The formalisms considered are Timed Automata and a super-class of these, called Hybrid Automata. We investigate their strengths and weaknesses and make a choice based on their ability to meet the requirements stated above.

- ***Timed Automata.*** The notion of Timed Automata, described in the introduction (see section 1.4.1), meets many of the requirements specified. They allow explicit modelling of real-time constraints by the use of *clocks*, which progress at a uniform rate. The state of the environment is modelled by a transition system with *locations* and *transitions*.

  In addition to the structural state modelling, local and global bounded discrete variables are supported in several contemporary model checking tools. The values of these variables can be updated when a transition is taken, whereas the clocks can only be *reset*, that is, their value is set to zero.

Structurally, Timed Automata fit the requirements of modelling sporadic updates by transitions. Modelling the continuous updates, on the other hand, is somewhat more of a challenge. Although variables are supported in Timed Automata, these variables are only updated when a transition is taken. Therefore, if a control program uses a polling scheme, the current valuation of a variable will never differ as long as the automaton is in the same location.

In some cases, the continuous updates that occurs over time can be modelled by adding more locations to the model. The value of a variable in a given location represents a set of values that can be considered equivalent to the control program. Transitions between the added locations must be enabled at certain points in time so that the updates occur at times when the control program polls the environment for that value. The updates of the variable then must set the value equal to that specified for a continuous update of the duration since the last update.

The disadvantage of the approach described above is that extra effort is required to add the locations needed. These locations must be based on an analysis of the communication pattern between the control program and the environment. Thus, if the timing of this interaction is altered, a modification of the environment model is required. Finally, some control programs poll their environments so often that the approach becomes inapplicable.

- ***Hybrid Automata.*** We now consider a formalism in which the continuous updates are modelled explicitly as so-called *flows*. The formalism, known as *Hybrid Automata*, consist of the following ([HHWT97]):

  - *Variables.* A set of variables with values in the real domain.
  - *Control Modes.* A named entity which describe the possible states of the automaton. At all times, one and only one mode in an automaton is *active*.

    In each mode, a *flow* expression describes the continuous updates of the set of variables over time. These expressions define the derivatives of the variables, where a derivative x' of a variable x describes the update performed on x for each unit of time passed.

    A mode also contains an *invariant* expression, representing one or more intervals in the real-valued domain for each variable. In order for a given mode to be active, the valuation of the variables must allow the invariant expressions for each variable to evaluate to true. In other words, the valuation of each variable must be within the specified intervals.

    Finally, an *initial condition* expression defines which modes can potentially be initially active. As with invariant expression, the initial conditions represent intervals in the real-valued domain, which the initial valuation of the variables must satisfy in order for the mode to be a potentially initial.
  - *Control Jumps.* A control jump represents a transition from one mode to another. It is represented by an edge between two modes, called the *source* and

target modes. A jump from the source mode to the target mode can be performed if the source mode is the active mode and the jump is *enabled*. The criteria for enablednes are described after the introduction of edge decorations.

The edges are decorated by so-called *jump conditions*, which are expressions in terms of the variables. Like invariant and initial condition expressions they restrict the set of valuations allowing a jump between the source and target modes to be performed.

In addition to jump conditions, edged are labelled with at most one *event*. These events are used for internal synchronisation between two or more automata. In order for a jump to be made in one automaton, another automaton must be in a mode from where a jump with the same event can be performed. The jumps must me performed synchronously, that is, without progression of time between the two jumps.

Three requirements must be met for a jump to be enabled. First, the guard expression of the jump must evaluate to true. Second, if the jump is decorated by an event, some automaton must be able to synchronise on the event. Finally, the invariant expression of the target mode must be satisfied in order for the mode to be allowed to become active.

Using Hybrid Automata, the continuous changes in the environment can be modelled explicitly. The flow expression of the modes perform the continuous changes, and the jumps constitute sporadic changes in the environment. The continuous updates are supported through the derivatives defined in flow conditions.

Hybrid Automata are often used for model-checking control programs. As is the case when model-checking Timed Automata, the behaviour of the environment must be modelled as automata in order to interact with the control program automaton during validation. This interaction is modelled using events – allowing the environment to react to actuations from the control program and vice versa.

In figure 4.2, two automata modelling the barrel environment presented in figure 4.1 are presented. Since we are only interested in describing the behaviour of the environment, no control program automaton has been specified. Therefore, an external entity is required in order to control the environment. This is clear from the fact that the automata cannot synchronise on the events `EV_START` and `EV_STOP`. In section 4.2.3, we will elaborate on this distinction between events.

The automata of figure 4.2 share a variable, *h*, denoting the water level of the barrel. The first automaton, modelling the pump, has 5 modes (see figure 4.2 (a)). They describe whether the pump is on, off, overheated, suspended due to an error, or on but without a flow of water. The initial mode is off, from where the automaton may enter the modes on and suspended. These jumps are triggered exclusively by the receipt of events from other entities.

If, for example, an `EV_START` event is received, the on mode becomes the active mode of the pump automaton. In this mode, the water level of the automata decreases by the rate given by the flow expression. The mode is left only when either an `EV_STOP`

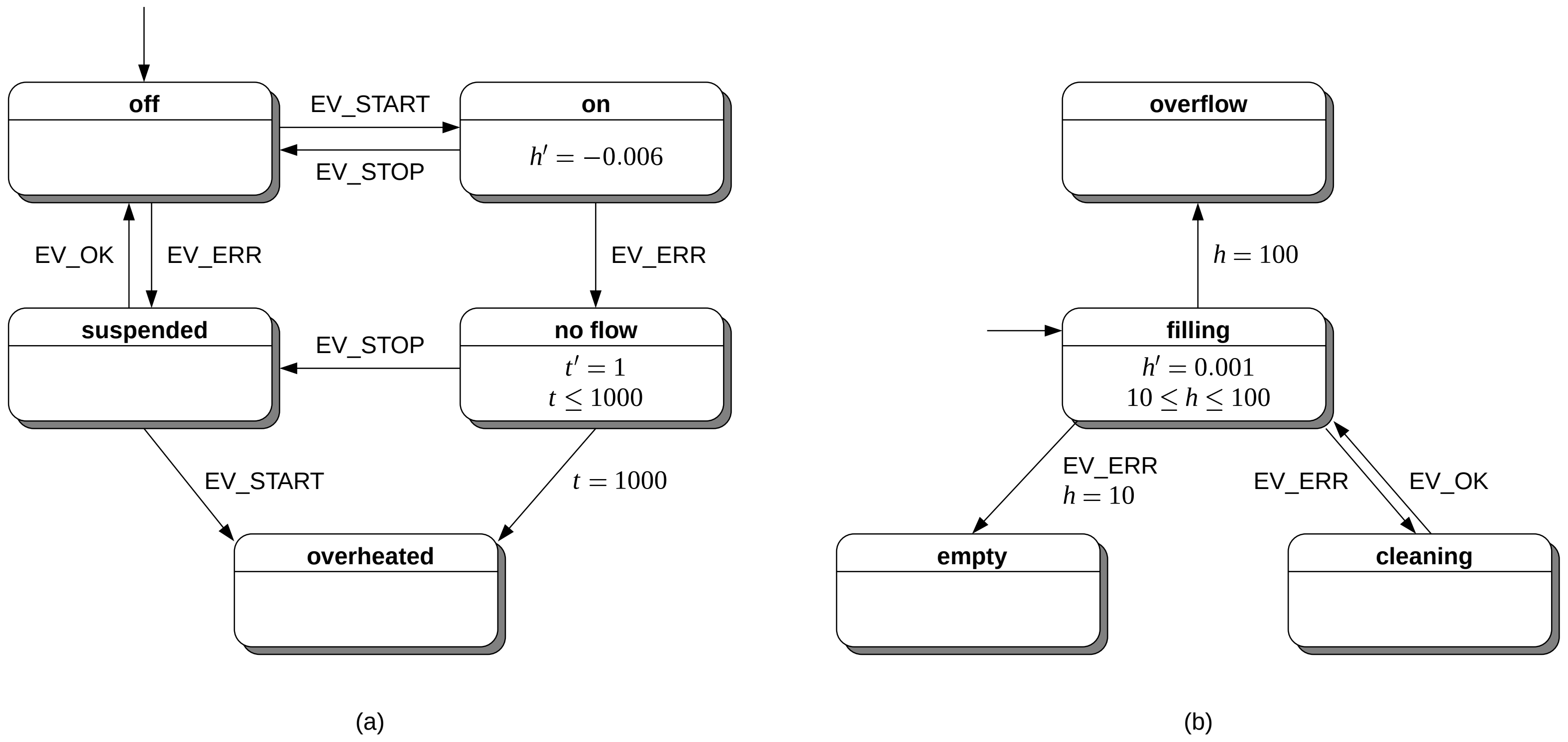

Figure 4.2: The environment of the barrel example modelled in Hybrid Automata

or and EV_ERR event is observed. In the latter case, the pump enters an intermediate mode, no flow, at which point the control program has one second to emit the EV_STOP signal. Otherwise, the pump enters its overheated mode, designating that the pump has been destroyed. This mode may also be entered if the control programs tries to start the pump while in the suspended mode.

The automaton modelling the barrel is given in figure 4.2 (b). Its initial mode is filling, where the water level of the barrel continuously raises. The water level must be between 10 and 100 units – otherwise the barrel is either considered empty or an overflow occurs. Finally, while filling, the EV_ERR event may be issued if the pipes are congested by dirt.

The two automata perform internal synchronisations on the EV_OK and EV_ERR signals. As earlier mentioned, the EV_START and EV_STOP signals are considered external, so the automata cannot synchronise on them.

Notice that the pump automaton may be in the on mode at the same time as the barrel automaton is in the filling mode. These both contain flow expressions involving the height of the water level. In such cases, the sum of the flow expressions is used – in this case yielding $h' = -0.005$. Also, in modes where the derivative defined on a variable is zero, we do not write it on the mode.

Since Hybrid Automata offer the best explicit modelling of sporadic and continuous changes, we choose to base our formal representations of environments on this formalism. We shall, however, introduce some modifications making them suitable for our simulation purpose.

4.2.2 Preliminary Simulation Considerations

Before describing the modifications of the Hybrid Automata, we shall make an initial choice of how to perform the simulation of the models. This decision influences the modifications necessary to the definition of Hybrid Automata. Thus, it is made before the modifications are introduced. A detailed description of the simulation approach is given in section 4.3.

In order to simulate a Hybrid Automata, it is necessary to maintain the current modes and variable valuations during run-time. Furthermore, the set of possible next states of the environment must be calculated from the flow and invariant expressions on the active mode as well as the guard expression of the possible transitions. For this, we consider two different solutions:

- *Interaction with a model-checker.* Using this approach, the model of the environment is maintained by a model-checker. An example of a model-checker for Hybrid Automata is *HyTech* ([HHWT97]). The set of next states can be calculated by the *reachability* algorithm applied by that model-checker.
- *Code generated from Hybrid Automata.* With this approach, the simulation is performed by running code. The flow, invariant, and guard expressions are given in terms of executable code, which manipulates a set of variables with a numeric type.

The main advantage of interacting with a model-checker is that an existing tool is applied. Thereby, the effort of implementing data structures to maintain the model and functionality to perform some of the required operations can be avoided. Using a model-checker does, however, impose restrictions on the complexity of the flow, invariant, and guard expressions allowed. For example, with generated code, these expression may involve the invocation of generic functions – which is not supported by current model-checkers.

We decide to perform simulation by generating executable code from the Hybrid Automata. We find that the added ability to express the environment is favourable compared to the extra effort required to implement the functionality for maintaining and simulating the models.

4.2.3 Modifications of Hybrid Automata for Simulation

In order to suit our purpose of simulation, we introduce some extensions and restrictions to the notion of Hybrid Automata. The extensions are mainly introduced in order to allow communication between the environment and control program during simulation. The restrictions are mainly introduced in order to limit the complexity of the simulation algorithm.

In order to allow the environment to interact by signalling (actuation and stimulation events alike), we categorise the events by which jumps can be decorated into two classes:

- *Internal events.* The internal events are used for synchronisation between automata in the usual manner, as described in section 4.2.1. For example, in figure 4.2, the `EV_ERR` and `EV_OK` events are internal to the model.
- *External events.* External events are used for interaction with the control program. The external events are further divided into the categories of *actuation events* and *stimulation events*. When a jump is decorated by an actuation event, this event must be received from the control program in order for the jump to be enabled. Stimulation events, on the other hand, do not influence whether a jump is enabled or not. However, when a jump decorated by a stimulation event is performed, the given event must be emitted to the control program. The `EV_START` and `EV_STOP` events of figure 4.2 are examples of external events.

The distinction between input and output events is well-known from the so-called Input/-Output Automata, used for model-based testing (see section 1.4.3). As a limitation to the project scope, we shall assume that the models of the environment do not involve internal synchronisation by the use of internal events.

When Hybrid Automata are used for model-checking, some restrictions are imposed on the flow expressions in order to be able to calculate the set of reachable next states. One such restrictions is that *the variables [...] evolve along a differentiable curve* ([HHWT97] pp. 4). For our simulation purposes, we shall disregard this requirement and introduce another.

We shall refer to the added requirement as *local invariance*. The requirement states that, for sufficiently small intervals of time, the evaluation of the invariant of a mode is consistent. Consider an invariant expression $i : \mathbb{R}^n \to \mathbb{B}$ and a non-negative integer, δ. Further, assume that two points in time, t_1 and t_2 where $t_2 - t_1 \leq \delta$, are given. If the invariant expression evaluates to the same boolean value, b, at t_1 and t_2, we say that flow expression is locally invariant in the interval $I = [t_1, t_2]$ if and only if for all $t \in I$, the invariant evaluates to b.

The added requirement to the flow and invariant expressions of a mode is that, for some given δ, all flow expressions must be locally invariant. The rationale for this restriction is that, during simulation, it is imperative to be able to find the amount of time (less than or equal to δ) an automaton can remain in its active mode. By the assumption of local invariance, this calculation can be performed using a binary search algorithm.

Since physical time is assumed never to stop, we shall require that a model of an environment always allows time to progress. In other words, a deadlock in the environment is not allowed to occur. If there is a chance that an environment may deadlock in case the control program misses an actuation, this must be modelled explicitly in the automata by adding a so-called *error mode*. For example, the overheated mode of the pump automaton given in figure 4.2 (a) is an error mode. If the mode was omitted, and the control program failed to send the `EV_STOP` signal within one second after the error was signalled, the environment would deadlock.

As a final restriction, we shall allow the automata to have only one initial mode. This mode is given explicitly by decorating it with an *initial* label. The variables are assigned default values, which must satisfy the invariant expression of the initial mode. Variables may also be assigned the *parameter* label, which means that the initial value of the variable can optionally be given as an argument.

Though the modifications above distinct the automata used for simulation from Hybrid Automata, we shall refer to them as Hybrid Automata throughout the remainder of this report. In the following, we present how the environment of the barrel example can be modelled as one of these modified automata.

Example

In figure 4.3, we present a hybrid automaton of the barrel environment with a distinction between input and output events. Events prefixed by the string "i:" are inputs from the control programs, that is, control program actuations. Events prefixed by "o:" represent the requirement to stimulate the control program by the given event.

Since internal synchronisation between automata is not supported, the automata for the pump and environment are merged into a single automaton. As a consequence, some of the modes of the earlier model are merged. For example, the pumping mode represents two modes, filling and on, in figure 4.2. Similarly, the repair represents the two modes suspended and cleaning.

Several automata for describing an environment are supported, however, if only they do not rely on internal synchronisation.

Notice that the overlap between guard expressions and invariants on their source modes have been relaxed. For example, transition from the filling mode to the overflow mode is allowed to occur in the interval between 99.9 and 100. This relaxation is introduced because the height of the water level is represented by a real-valued variable during simulation. Since the representation of this variable is not precise, it is necessary to allow the transition to occur within some interval.

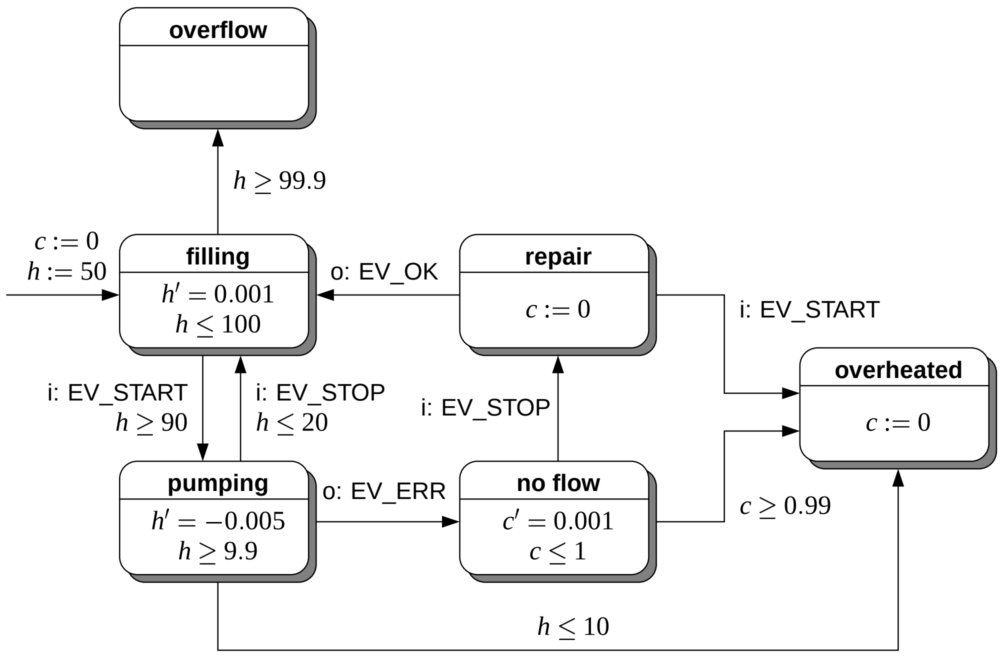

Figure 4.3: The environment of the barrel example modelled as a single modified Hybrid Automaton

4.3 Simulation

As earlier stated, the overall goal of formalising the description of the environment is to allow a simulation to performed. During this simulation, the state of the environment is observed, and, if an undesirable situation is reached, the control program is assumed to be incorrect. As described in section 4.2.2, we have chosen to perform this simulation by generating executable code from the environment models.

In this and the following sections, we shall describe how the simulation is performed. We first decide between two different simulation algorithms. The main difference between these approaches are the accuracy of the interaction with the control program. Afterwards, we describe how updates of the environment are performed. In section 4.4 we consider different strategies to *explore* the environment, that is, obtain a good coverage of its behaviour. Finally, in section 4.5, we explain how validation can be performed in terms of the states of the environment.

During runtime, the interaction between the environment and the control program requires the environment to accept actuations from the environment, emit stimuli events, and support the polling of the values of variables. Naturally, the update of the environment must occur concurrently with the execution of the control programs. The communication between the environment and control program must adhere to two requirements:

- *Actuations force jumps.* When a control program actuates on its surrounding, it is imperative that a reaction in the environment occurs if one is possible. Otherwise, an error that arises from a missing signal cannot be distinguished from an error arising from a *discarded* reaction. If, on the other hand, a discarding of a signal can in fact occur due to an error in a physical component, it must be explicitly modelled in the automaton describing the behaviour of that component.

- *No added delays when polling.* When the control program needs to poll the environment for the value of some entity, the delay from the time at which the poll request occurs to the value is returned should be no longer than the time it takes to read the value of the variable. If an unbounded delay could occur due to the polling of some entity, a deadline for the process performing the poll could potentially be broken. Specifically, this means that no overhead in delay must be imposed by the update of the environment. A *current* value of a variable must always be immediately available.

When the environment is performing an update, there will inevitably be a period of time in which the environment is in a state where it cannot be polled. Unfortunately, when the control program sends a request to poll the value of some variable, it is necessary to perform an update. Hence, the requirement that no delay must be imposed by polling cannot be met. In order to solve this problem, we represent the continuous evolution of the environment by discrete points. The state of the environment is only updated at certain points in time, and then remains fixed until the next update is performed.

The discrete representation of the environment allows the environment to work on a copy of the environment while the control program interacts with the most recent iteration. When the update has been performed, the resulting iteration of the environment is made the active one. This discrete representation introduces an inaccuracy in the values obtained by polling. However, given short update times, this inaccuracy can be disregarded. We therefore choose to operate with a periodic update of the environment, where the period of the update task is sufficiently long to perform the update while, at the same time, short enough not to introduce too much inaccuracy.

In chapter 5, we describe how the validation approach suggested in this chapter can be implemented in the inquisitor framework, presented in chapter 3. In this framework, logical time can be stopped while the update of an environment is performed. Thus, the length of the update does not constitute a problem, since the timing scopes of control program processes are not effected. Because of this, the issue of ensuring that the simulation period is sufficiently long to perform updates is not considered in the scope of this project.

In figure 4.4 (a), an example interaction between a simulated environment and a control program is given. After t_1 time units, the environment stimulates the control program by some event. It then waits for t_2 time units, unless an actuation from the environment triggers the need for an update. In the figure, such an actuation does in fact occur after t_3 time units ($t_3 < t_2$). With the periodic update approach (see figure 4.4 (b)), actuations and stimulation events must instead be buffered – since they cannot be handled at arbitrary times during an update.

Using the buffering solution described above, all stimuli emitted by the control program will occur when the active iteration of the environment is changed. When actuation events are received, they are stored along with their time of occurrence. The update procedure,

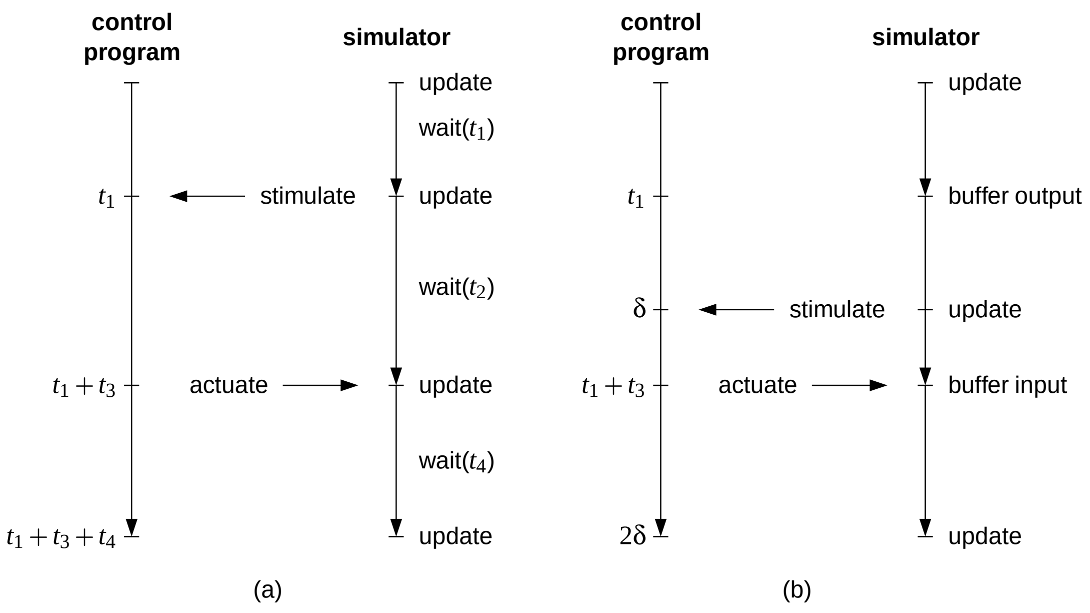

Figure 4.4: The aperiodic (a) and periodic (b) approaches to simulation

described in the following section, must comply to the requirement that actuations events force jumps if possible – and that the changes in the environment are performed at the time of their occurrence.

4.3.1 The Update Procedure

In this section, we describe how the periodic updates of the environment are performed. We shall refer to the period, δ, at which an update is performed as the *simulation period*. The update task performed periodically is referred to as an *update round*. At the beginning of the update round, a (possibly empty) set of time-stamped actuation events is available. At the end of a round, a (possibly empty) set of stimulation events must be emitted.

The task that must be performed during each update round is to find a way to distribute the δ time units passed since the last round. This is achieved by selection flow and jump actions in the environment. The simulation round is finished when the sum of the flows that have been performed equals the simulation period.

Jump, which are enabled by events present in the actuation event vector have precedence. Thus, if in at least one automaton a jump labelled by a actuation event *can* be taken, it *must* be. Furthermore, in order to comply with the requirement that actuation event must cause changes in environment state at the time of their occurrence, a flow in the current modes of the automata may be required.

By the approach described above, the changes in the environment caused by actuation events are registered as having occurred at the actual time. Thereby, during validation, the control program is never penalised by the fact that an update may not occur before δ time units after the emission of a signal.

4.4 Exploration Strategies

The previous section described how the formal specifications of an environment can be used to simulate the behaviour on an actual environment. In the simulation, some limitations are imposed by the interaction with control programs. However, a lot of *freedom* in the exploration of the model still remains. When an automaton is in some mode, and no jump decorated by an input event is enabled, there will often be a choice between performing a flow in the current mode, or performing a jump to another. If it is decided to stay, the length of the flow must be decided. If, on the hand, a jump is chosen, it must be decided which of the enabled jumps to pursue.

We refer to an algorithm for choosing whether to perform flows or jumps during simulation as an *exploration strategy*. Certainly, the behaviour of the environment during the simulation is dependent on the strategy employed. A good coverage of the behaviour of the environment is a precondition for a thorough validation of a control program. Thus, the aim of an exploration strategy is to lead to a good coverage of the environment model.

In the following, we describe a number of different exploration strategies that can be applied. The strategies are divided into two classes; Those that require historical data about the choices previously made, and those that do not:

- ***Non-historical strategies***

  - *Random.* A simple strategy is to always choose a random action. The advantage of this strategy is that all possible behaviours of an environment is obtainable. On the other hand, there is no guarantee that a good coverage is achieved.
  - *Jump and flow eager.* With these strategies, the simulation either tries to stay in the same mode as long as possible (flow eager), or performs a jump whenever possible (jump eager). The coverage of these strategies are good in terms of the continuous and sporadic changes in the environment. They are further characterised by their relatively high degree of determinism. This may be exploited for making special-purpose environments for testing specific behaviours of the environment.

- ***Historical strategies***

  - *Mode and jump coverage.* By storing information about the frequency by which each jump has been performed and how much time has been spent in each mode, a goal for an exploration strategy could be to try to obtain some equilibrium. For example, it could be desired that an equal amount of time is spent in all modes or that all jumps are performed an equal amount of time. Of course, since the signals received from the control program are uncontrollable, no guarantee can be made that this aim can be fulfilled.

    In the event that the testing team has a priori knowledge about behaviours that are likely to cause the control program to fail, *weights* on the modes and jumps can be used to guide the exploration performed by this strategy. For example, a low weight on mode is interpreted as a request to spend a short amount of time there, and conversely a high weight is interpreted as a request to stay longer. Similarly, a high weight on a jump should yield a higher probability that the jump is performed when enabled.

- *Value-based.* By observing the values taken by the variables of the automata during a simulation, choices could be made with the purpose of obtaining some value-coverage. For example, the domains of the variables could be partitioned, and a coverage aim could be that all partitions of all variables had been represented by some valuation of the variables during validation. For this strategy, a manual partitioning of the variables is required.

From the descriptions of exploration strategies above, it is clear that each strategy has strengths and weaknesses. Hence, no single strategy can be declared superior to all others. One solution to this problem could be to use a combination of strategies – either simultaneously or by changing the strategy during runtime. In order to allow a change between strategies, it is necessary to store historical information for all the strategies. This may induce a large overhead, which could considerably increase the time consumed by each simulation round.

4.5 Validation

As stated in the project scope, our main focus in this project is on simulating environments from formal descriptions. Consequently, the validation support described in the following section allows only relatively simple requirements to be stated and checked during runtime.

As is customary for runtime validation, we consider the happenings during an execution as a trace of events. In our case, the events of a trace consist of mode changes. The requirements for the system are formalised into properties expressing limitations in the allowed traces. Thus, a safety requirement can be expressed as the absence of entering some mode. Liveness requirements can be expressed by requiring some mode to eventually be entered.

As earlier mentioned, timing requirements for the control program are modelled as error modes in the environment. Thus, properties of control programs including time can be modelled simply by a safety property stating that a specific error mode is never reached. We have therefore chosen to focus on the implementation of safety properties.

For this purpose, we introduce a simple way to restrict the allowed traces. We shall refer to these restrictions as *mode constraints*. A mode constraint consists of two *mode properties*, which refer to a mode of some automaton and are satisfied if, during runtime, the active mode is the referenced mode. The general form of a mode constraint is:

If some mode property p_1 is satisfied **then** some mode property p_2 must be satisfied.

The above definition allows constraints on the modes of parallel compositions of automata. At any time during the simulation, one and only one mode is active in each automaton. Thus, in fact, the trace of an execution consists of a sequence of sets of active modes, referred to as *super* modes. A mode property expresses that some mode of a given automaton is active, whereas a mode constraint restricts the super modes allowed.

Design

CHAPTER 5

In this chapter we describe how the validation approach presented in the previous chapter has been implemented in the Inquisitor framework (presented in chapter 3). We start out by giving an overview of the validation process and the interaction between the runtime components. We then proceed to present how executable code is generated from Hybrid Automata and how the simulation and validation is performed. Finally, the status of the implementation and some suggested improvements are presented.

5.1 Overview of the validation process

An illustration of the validation approach is given in figure 5.1. The validation process consists of 4 stages; the informal, formal, executable, and conclusion stages. These stages and the transitions between them are described below:

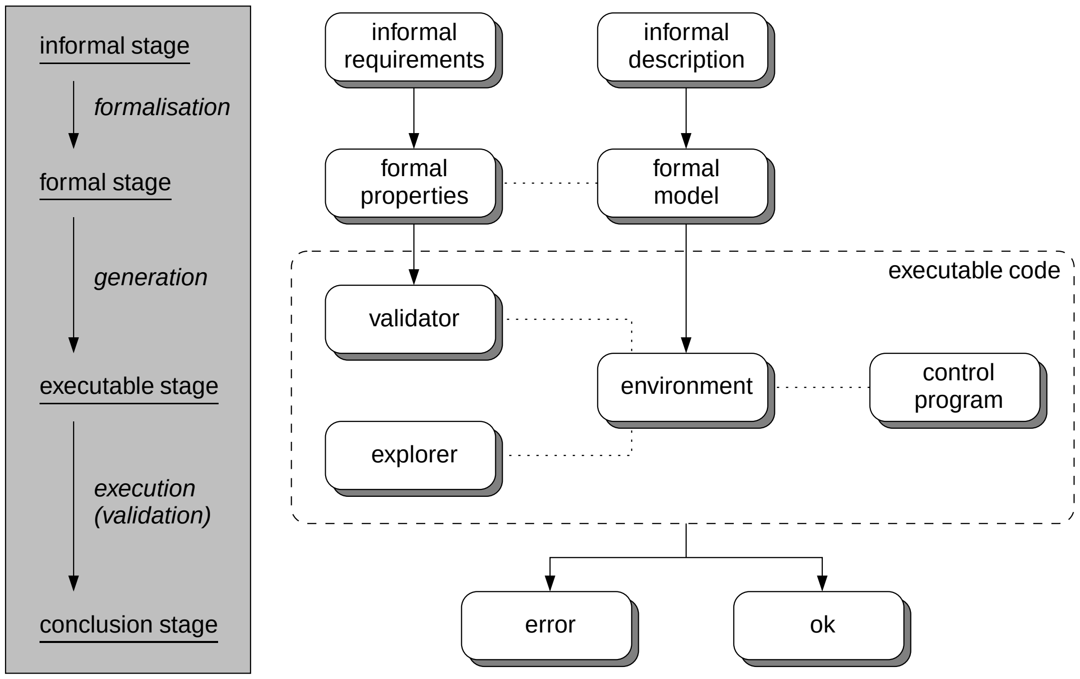

Figure 5.1: The validation process

- *The informal stage.* At this stage, only informal descriptions of the behaviour of the environment and the required properties exist. These are likely to be part of the requirement specification of the control program.
- *The formal stage.* By the process of formalisation, formal models and properties are created from the informal environment descriptions and requirements. The entities of the environment are modelled as Hybrid Automata, and the requirements are stated as mode constraint properties. The formalisation process is carried out manually by the validation team.
- *The executable stage.* From the formal representations of the environment and the properties, executable code is generated. The code generated from the hybrid automata is used for simulating the environment during the validation. In addition to interacting with the control program, the simulated environment interacts with an *explorer* component, which employs one of the exploration strategies presented in section 4.4. The code generated from the set of properties defines a set of mode constraints that are checked by the *validator* component.
- *The conclusion stage.* For the actual validation, the generated code is executed with the two possible outcomes that an error was found or than no error was found. To emphasise that it cannot be concluded that no error is contained in the control program though none was found, we have called the conclusion state *ok* instead of *no errors*. For similar reasons, this conclusion can only be obtained by terminating the validation process – either by a timeout or from some coverage requirement.

The actual validation is carried out at runtime by executing the control program in parallel with the simulated environment. In the next section, we describe the responsibilities of each of the component present at runtime (at the executable stage) and the interaction between them.

5.2 Component Architecture

The component architecture at the executable stage described above is repeated in figure 5.2, decorated with labels describing the interaction between them. Each of the components (including the control program) are active entities. The environment, validator, and explorer execute as inquisitors, since the time they spend executing should not influence the timing scopes in the control program. In the following section we describe the responsibilities of each component.

5.2.1 Component Responsibilities

The central component is the environment, which is generated from the Hybrid Automata specification. As will be clear from the description of this generation in section 5.3, the structure of the automata will be maintained in the generated code. The main responsibility of the environment component is to act as data for the other components and perform the

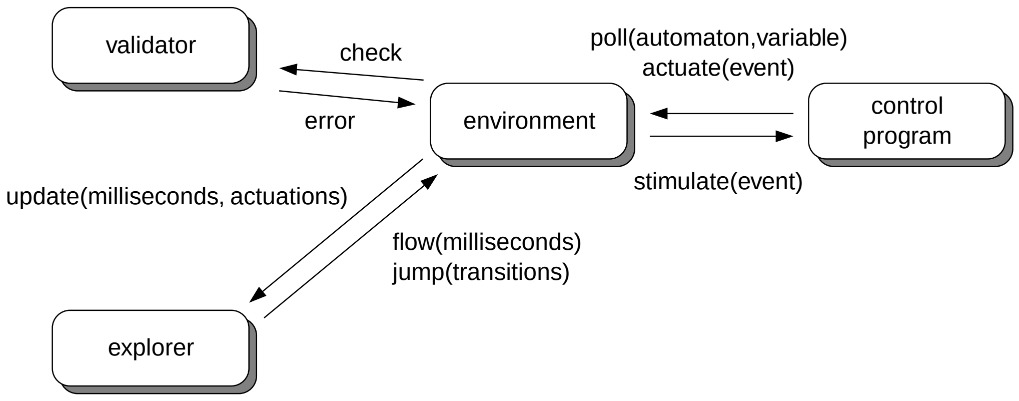

Figure 5.2: The interaction between the components

actions they dictate. The environment component is also responsible for initiating each simulation round periodically.

The actions dictated by other components include receiving the events actuated by the control program and acting as a pseudo sensor entity that the control programs can poll. The events from the control program must be time-stamped in order to be effectuated at the right time during the following simulation round. It also involves performing the flow and jump updates chosen by the explorer – possibly involving stimulation of the control program. The responsibility of the explorer component will be elaborated on later in this section. Finally, the environment must act and report on errors found by the validator component.

The responsibility of the validator component is to monitor the state of the environment – ensuring that all properties are satisfied. These checks are performed at the request of the environment each time a change in the environment has occurred. Notice that, as a consequence of the fact that updates of the environment can only be performed by the environment itself (not directly by the control program), these checks are performed only during the simulation rounds.

Each of the properties have a textual description. If a property is found not to be satisfied during simulation, this is reported back to the environment. The environment then reports the error and, in the current implementation, terminates the execution with an *error* conclusion state.

The explorer has the responsibility of choosing the updates that the environment carries out. At the start of each simulation round, the environment informs the explorer to choose the updates to perform. For this purpose, a vector of the actuations received during the simulation period is given as well the maximum time allowed to flow. An update can consist of either a flow, a jump, or a flow followed by a jump in case a transition decorated with an actuation event is enabled.

If a jump or a flow shorter than the remaining simulation time is chosen, the environment must re-inform the explorer to choose another update. This procedure is repeated until the simulation round is over – that is, the sum of all flows equals length of the simulation period.

During this procedure the validator performs a check each time an update has occurred. Furthermore, any stimulation events are buffered by the environment and emitted at the end of the round.

5.2.2 Component Interaction during Simulation

Having defined the responsibilities of the components and the interaction between the environment and each of the other components, we proceed to describe the order of the interaction during the simulation procedure. The environment component controls the explorer and validator components. That is, these components only become schedulable after a notification from the environments. When they have performed their tasks they notify the environment and wait for notification.

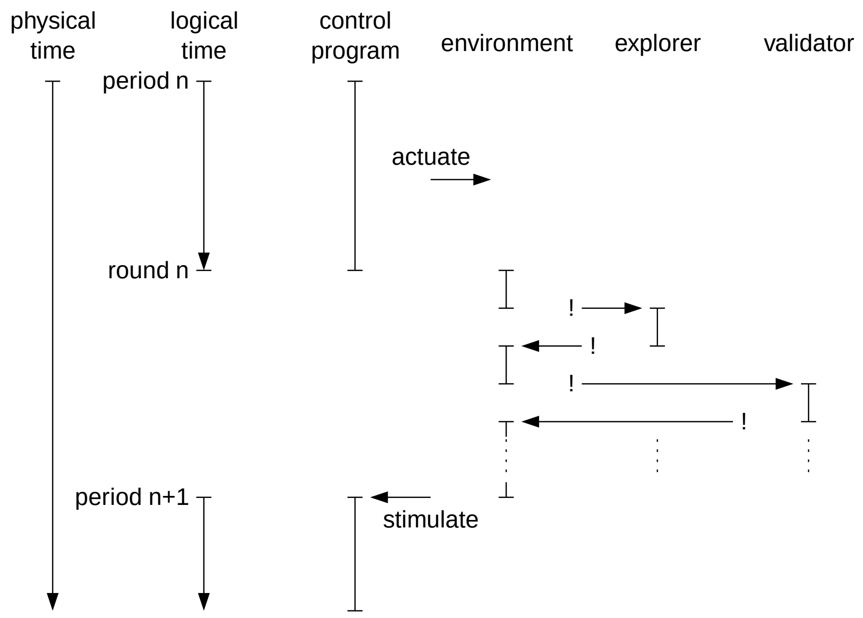

Figure 5.3: The order of the interaction between components during simulation

The components execute in the following order (see figure 5.3):

1. *Control program execution.* Between each simulation round, the control program executes. During the execution, the control program may actuate on the environment. As earlier mentioned, these actuation events are time-stamped and stored in the environment.

2. *A simulation round.* The control program executes until the periodic environment thread becomes schedulable. Since the environment is an inquisitor thread, it has the highest priority and, consequently, the control program is preempted. At this point, logical time stops since an inquisitor thread becomes running. The environment must now be updated with a maximum flow, δ, equal to the simulation period. The

simulation round is carried out by repeating the following algorithm until δ equals zero:

(a) Notify the explorer to perform an update spending at most δ milliseconds;

(b) Wait for the explorer to notify;

(c) Perform the flow and/or jumps specified by the explorer. Buffer the stimuli events associated with jumps;

(d) Notify the validator to check whether all the properties are satisfied;

(e) Wait for the validator to notify and report an error if a property is not satisfied;

(f) Subtract the flow-time of the update performed from δ. If no flow was performed, δ remains unchanged.

(g) If δ is larger than zero, repeat the above procedure.

3. *Emitting signals.* After the simulation round, the buffered stimuli events are emitted. After this, no inquisition threads are schedulable, so the control program again becomes running and logical time starts to progress.

In the following sections we present the design of the three inquisition components, the environment, the explorer, and the validator.

5.3 The Environment Component

As described in section 5.1, the environment is generated from hybrid automata and acts as data for the other components. The component's thread is responsible for notifying the explorer and validator components when they should execute, perform the updates chosen by the explorer, and inform about errors reported by the validator.

In this section we shall first describe how the Hybrid Automata models are specified by textual descriptions. We then give an overview of how executable code is generated from these descriptions, and explain how simulation and validation is supported in the environment component.

5.3.1 Model Specification

The Hybrid Automata describing the behaviour of the environment are specified in the eXtensible Mark-up Language (XML) format. The structure of an environment description is given by a document type definition (see appendix A.1).

We have chosen to describe the specification of an environment by an example. For this purpose, reconsider the barrel example given in section 4.2.3. A partial description of this system is given in figure 5.4.

The environment consists of number of events, automata and instantiations. In the barrel example, the environment is named "BarrelEnvironment", and there are two input and two output events, named "EV_START", "EV_STOP" etc. In the example, only one automaton, "Barrel", is defined. This barrel automaton has a single variable, "height" with a default

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE environment SYSTEM "eha.dtd">
<environment name="BarrelEnvironment">
  <event label="EV_START" type="INPUT"/>
  <event label="EV_STOP" type="INPUT"/>
  <event label="EV_ERR" type="OUTPUT"/>
  <event label="EV_OK" type="OUTPUT"/>
  <automaton name="Barrel" dynamic="false">
    <var name="height" defaultval="0" parameter="true"/>
    <mode name="filling" initial="true">
      <update>me.height = me.height + millis * 0.001;</update>
      <invariant>me.height &lt;= 100</invariant>
    </mode>
    <mode name="pumping" initial="false">
      <update>me.height = me.height - millis * 0.005;</update>
      <invariant>me.height &gt; 9.9</invariant>
    </mode>
    <mode name="overflow" initial="false">
      <invariant>true</invariant>
    </mode>
    <transition>
      <source mode="filling"/>
      <target mode="pumping"/>
      <guard>me.height &gt;=90</guard>
      <weight>1</weight>
      <input>EV_START</input>
    </transition>
    <transition>
      <source mode="pumping"/>
      <target mode="filling"/>
      <guard>me.height &lt; 20</guard>
      <weight>1</weight>
      <input>EV_STOP</input>
    </transition>
    <transition>
      <source mode="filling"/>
      <target mode="overflow"/>
      <guard>me.height &gt;= 99.9</guard>
      <weight>1</weight>
    </transition>
  </automaton>
  <instantiation>Barrel(50)</instantiation>
</environment>
```

Figure 5.4: Partial textual description of the environment presented in section 4.2.3

value of 0. The variable is a parameter since it has the parameter attribute set to "true". This means an initial value for the variable must be given as an argument when an automaton is instantiated.

The automaton further consist of a number of modes and transition. Each mode has a unique name and a parameter specifying whether the mode is the initial mode of the automaton in which it is defined or not. Furthermore, Java expressions for flows (named updates in the XML description) and invariants are defined. Note that, whereas invariant expressions are required, flow expressions are not. For example, no flow expression is defined in the overflow mode. The rationale for this will be explained in section 5.3.2.

Transitions consist of a source mode and a target mode, both referenced by their unique names. They also include a weight for use with exploration algorithms requiring them (see section 4.4). Furthermore, they contain a guard given by a Java expression and possibly a stimulation or actuation event – but not both (see section 4.2.3). Note that stimulation events are called inputs in the specification and actuations are called outputs.

The final definition in the environment is the instantiation of the automata described above. With the current implementation it is only possible to have one instance of each automaton (see section 5.6). In the example, an instance of the barrel automaton is created – with the initial value of the height variable set to 50.

5.3.2 Code Generation

Given a textual description of an environment, the next step is to generate executable code to simulate that environment. During simulation the environment must perform two tasks: One, interact with other components, and two, update itself according to the expressions associated with modes and jumps. The basic interaction tasks, such as receiving actuation events from the environment and informing the explorer to perform a simulation round do not differ from one environment to another. Neither does the structural relations between an environment and an automaton, and in turn, the automatons modes and transitions.

When generating executable code from the Hybrid Automata we take advantage of the similarities of the models described above. Each of the entities of a Hybrid Automaton are mapped on to a Java class, as illustrated in figure 5.5. The `Environment` class extends the `Inquisitor` class in order to ensure that logical time does not progress when updates of the environment are performed. The Environment consists of a list of instances of the `Automata` class as well as a list of `Event` instances. The `Automaton` class holds a list of `Mode` instances which in turn holds a list of outgoing transitions. To each `Transition` instance, a source and a target mode are associated. Furthermore, an event (stimulation or actuation depending on the type given in the `Event` instance) and a weight field is included in the `Transition`

Although much of the underlying structure is identical to all models, the variables, events, modes (including flow and invariant expressions), and jumps (including guard expressions) are not. These model-specific parts are implemented by specialising the classes of the environment (see figure 5.5). In fact, most of these classes are declared *abstract* (those whose names are given in italics), since it does not make sense to instantiate the base classes.

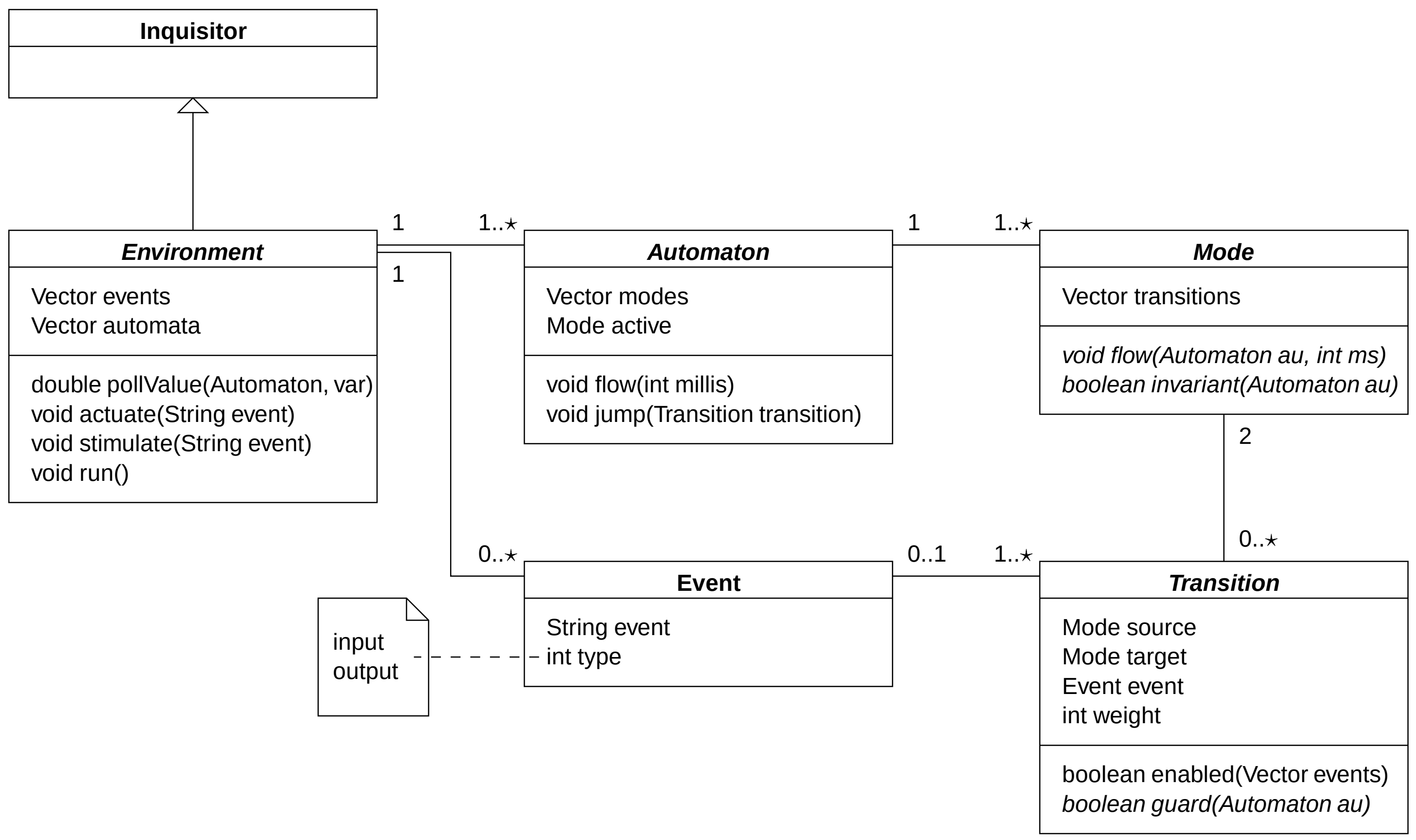

Figure 5.5: Diagram of the target classes of the code generation

The code generated performs the task of defining descendants of the abstract classes and instantiate them. In the following, we shall explain how this works by use of the barrel example for which a textual description in XML was given in figure 5.4. A complete listing of the generated code is given in section A.2.

A descendant for the `Environment` class, called `BarrelEnvironment` is generated. This class performs the task of instantiating the events defined, as well as the Barrel object. Next, code for the `Barrel` class is generated. This class specialises the `Automaton` class with a field named `height` of type `double` (that is, a double precision real-valued number). Furthermore, for each mode and transition specified in the automaton definition, an instance of an *anonymous* class are generated.

In the case of modes, the anonymous class generated contains two methods, `flow` and `invariant`. The `flow` method updates the variables of an automaton passed by argument according to the expression given in the textual specification. As earlier mentioned, this update need not occur in all modes. The `invariant` methods, on the other hand, must contain some expression that evaluates to either true or false – corresponding to the `boolean` return value of its declaration (see 5.5). Similarly, the `guard` method of the anonymous descendant of `Transition` returns a boolean value designating whether the guard is satisfied for the automaton passed by argument or not.

5.3.3 Simulation and Validation Support

Regarding interaction with other components during runtime, the various classes of the environment offer methods that fulfil the purpose. In the `Environment` class, the `PollValue( String automaton, String variable)` method allows the control program to poll the value of some variable in an automaton for its current value. The control program may also invoke the `stimulate(String event)` method, thereby actuating the given event on the environment. Analogously, the environment (which is an active object) may invoke the `stimulate(String event)` method in order to give the control program some stimuli.

The `run()` method of the environment performs the interaction (synchronisation) with the instances of the `Explorer` and `Validator` classes. These two classes specialise the `Inquisitor` class and are therefore active objects. The tasks they perform will be described in sections 5.4 and 5.5, respectively.

Once the explorer component has chosen whether to perform a flow, a jump, or both, the environment invokes the `flow(int ms)` or `jump(Transition transition)` methods, defined on the abstract class `Automaton`, on each of the automaton instances. After doing so, it notifies the validator to perform the validation task and reports any errors returned.

5.4 The Exploration Component

The active object of the exploration component is the abstract class `Explorer` (see figure 5.6). The run method of this class participates in synchronisation with the `Environment` instance. In order to choose an action for the environment to perform, a descendant of the `Explorer` class must implement the `choose()` method. This method typically employs one of the exploration strategies described in section 4.4.

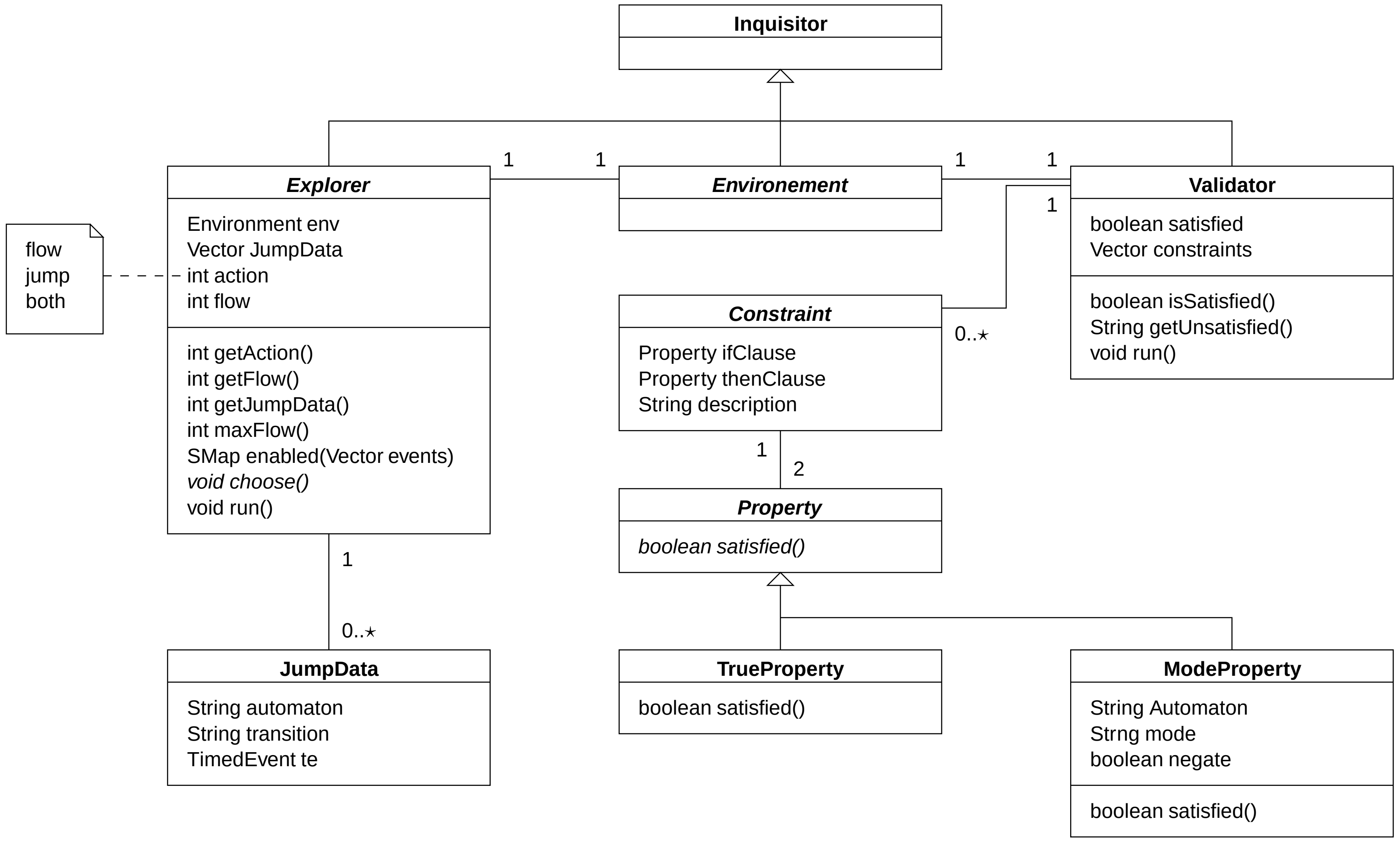

Figure 5.6: Class diagram relating the validation, environment, and exploration components.

In order to help this decision, two methods, `SortedMap enabled(Vector events)` and `int maxFlow()` are made available by the abstract class. The `maxFlow()` method investigates the current configuration of the automata and returns the longest amount of milliseconds that all automata can flow in their current modes. By the assumption of local invariance (see section 4.2.3) this can be calculated using a binary search approach.

The `SortedMap enabled(Vector events)` returns a set of `JumpData` instances, describing which transitions in which automata can be taken. Since jumps involving actuation events from the control program must be performed at the time of the occurrence of the actuation, they may be associated with an instance of `TimedEvent`. Class `TimedEvent` specialises class `Event` with a `RelativeTime` instance designating the time of the occurrence relative to the simulation period. The `JumpData` are sorted ascendingly by their time of occurrence. Instances without an associated `TimedEvent` are inserted at the end of the set.

The `choose()` method must set the desired update action to either flow, jump, or both. If a flow is selected, the `getFlow()` method must return the length of the flow. If a jump is selected, the `getJumpData()` method must return a list of `JumpData` objects instructing the environment thread which jumps to take. Furthermore, if the explorer cannot find a valid update it must enter a special deadlock state, not illustrated in the figure.

5.5 Validation

In this section we describe how the mode constraints presented in section 4.5 are implemented as Java classes, and how they are checked by the validator. The main class of the validator component is the inquisitor descendant, `Validator` (see figure 5.6). The `void run()` method of this class interacts with the environment – performing checks of all constraints when notified by the environment. When execution returns to the `Environment` thread, the `boolean isSatisfied()` method must return true if and only if all constraints are satisfied. If one or more constraints are not satisfied, the `String getUnsatisfied()` method can be invoked to get a textual description of the errors encountered.

Constraints are represented by instances of the class `Constraint`. This class encapsulates the if and then clause fields, represented by instances of the `Property` interface. The latter interface has a single method, `boolean satisfied()` that must return a value corresponding to the validity of the property. The simplest class implementing this interface is `TrueProperty`, whose `satisfied()` method always returns `true`.

For specifying that an automaton is either in a given state or not, the `ModeProperty` class can be used. In this class, the `String automaton` and `String mode` fields describe which state of a given automaton the property refers to. If the `negate` field is set to `false`, the `satisfied()` method returns `true` if an only if the automata is in the given mode. If, on the other hand, the value of `negate` is `true`, the method returns `true` if and only if the automaton is currently not in the specified mode.

When checking whether a constraint is satisfied, the `Validator` checks if the property field `ifClause` is satisfied. If it is, the property referenced by the `thenClause` field must also be. If the latter property is not satisfied, the constraint is broken, and an error must be reported. If, on the other hand, the `ifClause` is not satisfied, no requirements for the `thenClause` exist.

5.6 Implementation Status

The current implementation employs a simple exploration strategy, specifically, that by which the explorer chooses to stay in each mode as long as possible. This strategy was implemented due to its deterministic behaviour. In the next chapter, we carry out a case study, which, among other purposes, serves as a test of the implemented functionality. The deterministic behaviour of the chosen exploration strategy facilitates easier debugging than strategies with a more nondeterministic behaviour.

some limitations in the current implementation should be noted. One of them is, that no generation from a textual description of the mode constraints used for validation is available. The implication is that the constraints instances must me instantiated by manually written Java code. Also, since the inquisitor framework does not support the automatic instrumentation of Java byte-code, the extra code must be manually inserted.

Case Study CHAPTER 6

In this chapter we shall perform a case study in which we model an environment as Hybrid Automata and execute the generated code in parallel with a control program while observing a number of properties. This case study is conducted for three reasons. One, to exemplify the validation method described in the previous chapter, and second to serve as a test of the implemented parts of it. Finally, we hope to identify some strengths and weaknesses of the approach from this practical work.

The case we shall study involves a bridge over Oddesund in the north-western Jutland, Denmark. We first describe the components and operations of the bridge. We then model the environment as Hybrid Automata and state the properties in terms of the modes of these automata. Next, we construct a control program that performs the operations of the bridge. For testing, we introduce errors into this control program and execute it in parallel with the simulated environment. We then hope to be able to detect these errors during runtime.

6.1 The Bridge over Oddesund

The bridge over Oddesund is a so-called *bascule bridge*. It opens by sinking a counterpoise, thereby lifting the footway into the air. This allows ships to pass under the bridge. The bridge is depicted in figure 6.1. In addition to ships passing under the bridge, cars and trains cross the bridge. Thus, a system for holding back the cars and trains when the bridge is up is necessary. It is this logic that the control program of this case study will perform. We shall refer to the entity controlling the operations at the bridge as the *controller* whether consisting of a control program, or, as is currently the case, a human operator.

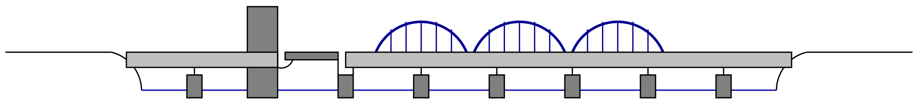

Figure 6.1: The Bridge over Oddesund.

The components of the bridge that we shall consider part of the system are:

- *A bridge bascule.* The bridge opens and closes by raising and lowering the bascule, respectively;
- *A set of rails.* The trains pass the bridge via a single set of rails. Ensuring that only one train is on the rails at the same time is performed externally to the environment described here;

- *A set of barriers stopping road traffic.* Before the bridge can start moving toward its upward position, a set of barriers located on the bridge on either side of the bascule are lowered in order to stop cars from passing the bridge;
- *Control lights for the trains.* Control lights for trains passing the bridge are placed some distance before the bridge. These tell the train controller whether to stop or pass the bridge;
- *Interaction with ships.* When a ship approaches the bridge, it must issue a request for the bridge to be opened. When allowed to pass, the bridge is opened and a grant for passing is sent to the ship. With the current operation of the bridge, this communication is performed via radio.

It should be noted that the above description is only partial. Among the parts excluded is an extra set of barriers used to stop traffic from one direction when a wide load must pass the narrow bridge from the other side. Furthermore, a large number of safety relays are used for ensuring that a flow of current exist in all electrical components in the surrounding. These include the barriers and warning lights on and before the barriers. The relays trigger alarms when a malfunction is observed.

We find, however, that the parts considered above constitute an appropriately complex system for our testing purposes. As further restrictions, we shall assume that trains only arrive from one direction, and that ships only approach the bridge from one side. In the following section we shall describe the operation of the bridge, serving as a refinement of the environment and a requirement specification of the control program.

6.2 Operation of the Bridge

In this section we turn our attention to how the bridge is operated. Specifically, we describe how incoming trains and ships are handled by the controller of the bridge, trains, and ships. This includes the communication between them as well as the actions they perform. An aerial overview of the entities of the environment is given in figure 6.2 along with distances from sensors and signals to the bridge.

6.2.1 Incoming Trains

An incoming train is detected by the controller with the aid of a sensor placed so that the trains position can be detected when it is between two and eight kilometers before the bridge. This sensor can be polled in order to obtain the train's distance to the bridge. By the time the train passes the signal 1.5 kilometers from the bridge, the bridge controller must have switched on the signal in order for the train to pass the bridge. Otherwise the train will stop 500 meters before the bridge waiting for the signal. The train travels at 80 kilometers per hour until the 500 meter mark from which it travels at 50 kilometers per hour.

Once the train has passed the bridge, it enters a sensoring zone stretching from 200 to 500 meters after the bridge. When the controller detects that the train has entered this zone, it may be concluded that the train has passed the bridge. After this point, the train may increase its speed to 80 kilometers per hour.

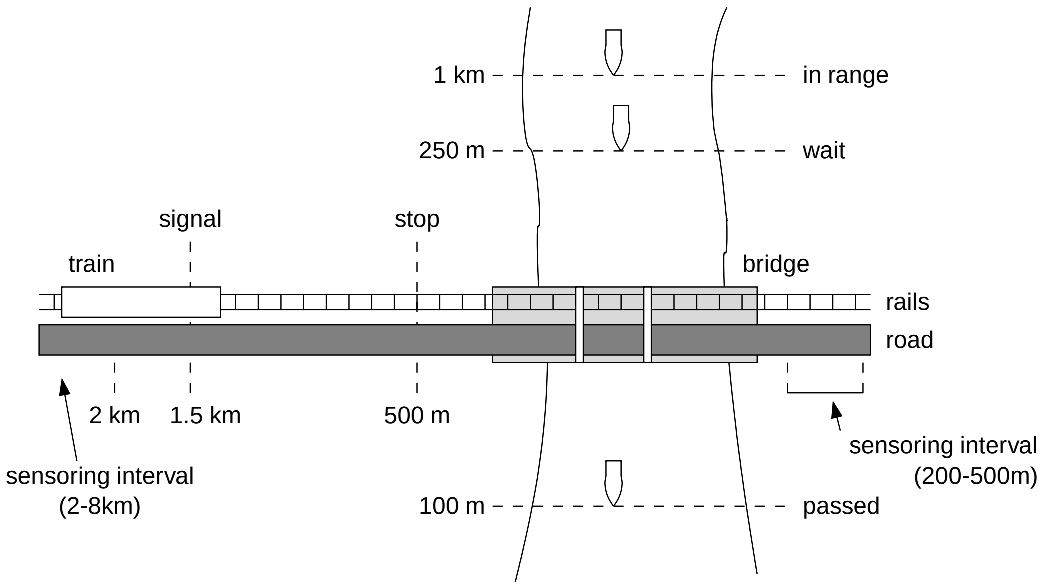

Figure 6.2: An aerial view of the bridge parts

6.2.2 Incoming Ships

When a ship is within 1 kilometer of the bridge, it enters the communication range. At this point, it must contact the bridge controller by issuing a request for opening the bridge. The bridge controller must respond with either a grant or deny of this request. In case a deny is issued, the ship must wait 250 meters from the bridge for a grant to pass under the bridge.

We assume that a ship approaches the bridge with a velocity between 3 and 5 meters per second when between the 1 kilometer and the 250 meter marks. From the latter mark it proceeds to pass the bridge at between 2 and 3 meters per second. When the ship is 100 meters on the other side of the bridge it issues a signal to the bridge controller that it has passed the bridge.

6.2.3 Opening the Bridge

The bridge must be opened and closed in correspondence with the signals given to the ships. Before a grant signal is sent to the ship, the bridge must be open. Similarly, the bridge is not allowed to close before the ship has issued its passed signal. Furthermore, the bridge must send signals in order for the barriers to lower and raise. We assume that the barriers raise and lower within 10 seconds. We also assume the presence of sensor that send signals when the bridge is fully opened and closed respectively. The procedure of opening and closing the bridge is given in table 6.1 below.

6.3 Modelling the Environment

In this section we describe how we have modelled the environment described above as Hybrid Automata. These automata will be used for stating the properties that we wish to

	Description	Done
1	Lower the barriers	after 10 seconds
2	Open the bridge	when signal received
3	Send a grant signal to the ship	immediately
4	Wait for passed signal	when signal received
5	Close the bridge	when signal received
6	Raise the barriers	after 10 seconds

Table 6.1: The procedure of opening the bridge

hold in the next section. Furthermore, code generated from these models will be used for the validation of the erroneous control programs in section 6.5.

The environment modelling consists of four Hybrid Automata representing the bridge, the barriers, the trains, and the ships. We describe these automata individually. In this section, the input and output signals given in the figures are relative to the environment. Thus, an input in the figure corresponds to an actuation signal by the control program. The textual description is listed in section A.3.

6.3.1 The Bridge Hybrid Automaton

The Hybrid Automaton for the bridge is depicted in figure 6.3. It consists of the four modes down, raising, up, and lowering. A variable, `deg`, represents the angle of the bridge bascule from its closed position. The bridge is initially closed. Thus, the `deg` variable is set to 0 and the initial mode is down. The bridge automaton remains in this position until it receives an `EV_BRIDGE_RAISE` input signal from the control program. It then jumps to its raising mode, at which the angle from the closed position increases until at most 80 degrees.

When the angle is between 79 and 80 degrees, a jump to the up mode must be carried out. As mentioned in the previous section, the bridge will thereby stop raising, and the `EV_BRIDGE_UP` signal is offered to the control program. Similarly, when an `EV_ BRIDGE_ LOWER` signal is received, the bridge starts to lower. This proceeds until the angle is between zero and one degrees. At this point the `EV_BRIDGE_DOWN` stimuli is emitted.

6.3.2 The Barrier Hybrid Automaton

Although the actual environment consists of four barriers closing the road for traffic from both directions, we shall assume that these behave identically and thereby can be modelled as a single automaton. This automaton, presented in figure 6.4, is identical in structure to the bridge automaton presented above. Like the bridge automaton the barrier automaton has four modes, denoting its current position and a single variable, `deg`, representing its degree relative to the initial position, up.

Furthermore, it reacts to actuation signals (`EV_BAR_LOWER` and `EV_BAR_RAISE`) from the control program for jumping between the non-moving (up and down) and moving modes (lowering and raising). The major difference lies in the fact that no signals are emitted when the barrier jumps from the moving modes to the non-moving ones. The rationale for this

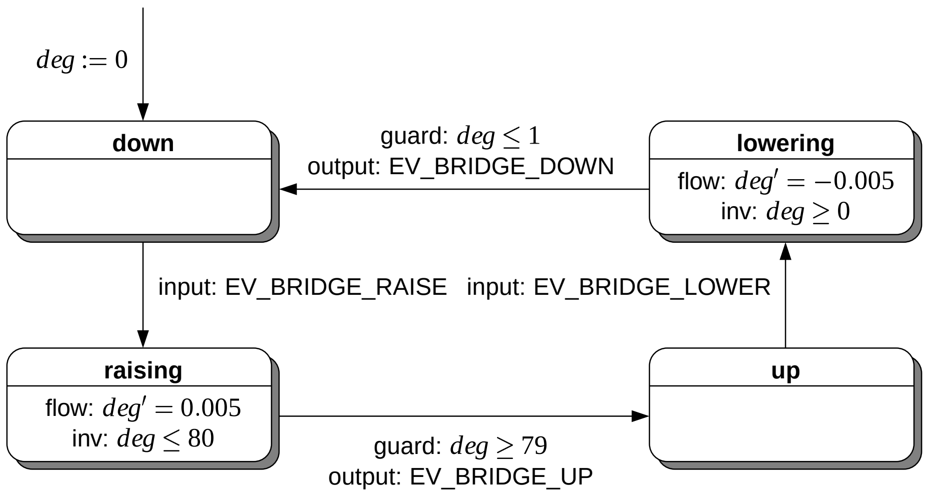

Figure 6.3: The bridge Hybrid Automaton

is that we wish to introduce the need for a timer in the control program. The barriers will spend 9 seconds on closing and opening, which allows the control program to conclude that the barriers are either up or down within the 10 seconds from the signals, as specified in the previous section.

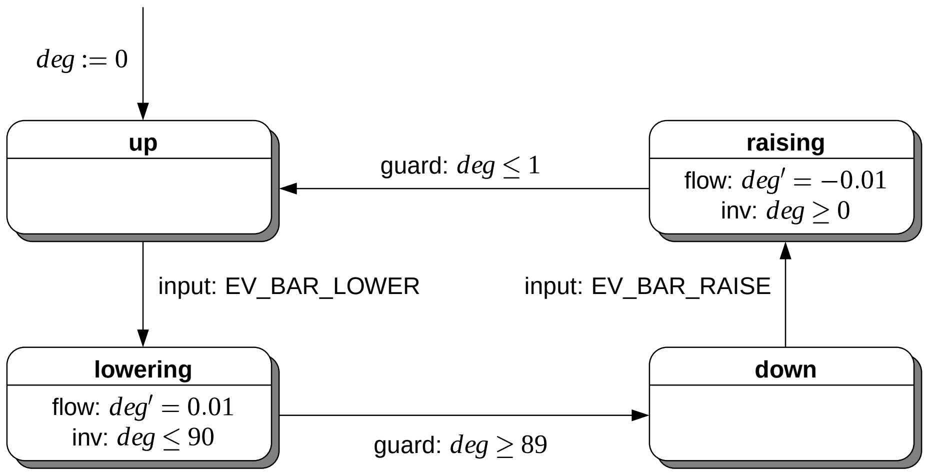

Figure 6.4: The barrier Hybrid Automaton

6.3.3 The Train Hybrid Automaton

According to the specification of incoming trains given in section 6.2.1, the control program can observe a train no sooner than when it reaches a sensoring zone 8 kilometers before the bridge, and no later than when it passes the sensoring zone 200 meters after. Therefore, the trains are sporadic entities which could be modelled using dynamic creation. We have chosen not to do so, primarily because the current implementation does not allow for dynamic creation. A second reason is that we assume the presence of at most one train at a time – whereby the advantages of dynamic creation is limited.

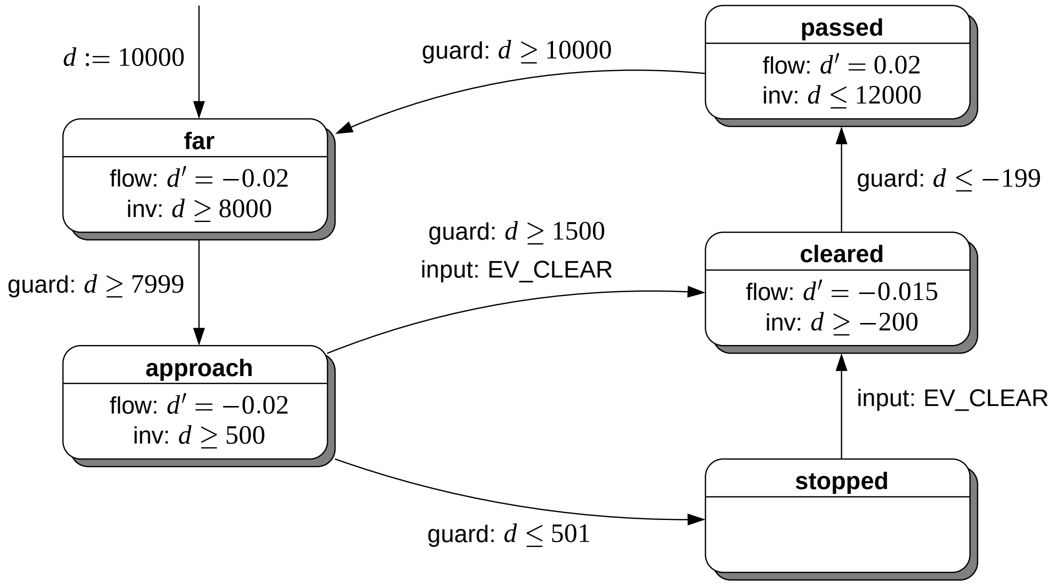

Figure 6.5: The train Hybrid Automaton

The Hybrid Automaton for the incoming trains is depicted in figure 6.5. The automaton contains five modes: far, approach, cleared, stopped, and passed. A variable, `d`, denotes the distance to the bridge. The initial mode is far, in which the train remains until eight kilometers from the bridge. A jump is then made to the approach mode, from which a jump to the cleared mode can be made if and only if a `EV_CLEAR` signal is received from the control program before the train passes the 1.5 kilometer mark. If this is not the case, the train will be forced to stop 500 meters before the bridge, waiting for the `EV_CLEAR` signal.

Once the train is cleared to pass the bridge, it proceeds at a reduced speed until 200 meters after the bridge, at which point it enters the passed mode. If we had chosen to use dynamic creation for trains, this would have been the final mode for the train automata. However, in order to allow more than one train to ever pass the bridge without dynamic creation, we introduce a cycle in the automaton. The train will remain in the passed mode until between 10 and 12 kilometers away from the bridge, at which it re-enters its far mode.

6.3.4 The Ship Hybrid Automaton

Like the trains, ships could be naturally modelled as sporadic entities. However, for the same reasons we did not model incoming trains using dynamic creation, we shall not use dynamic creation in the modelling of ships. The single automaton for modelling ships is shown in figure 6.6.

As in the train automaton, the ship's distance to the bridge is represented by a variable, `d`. Initially, the ship is in the far mode, in which it stays until the distance is 1000 meters. It then takes a jump to the in range mode, whereby a `EV_IN_RANGE` stimuli is emitted. The automaton stays in this mode until either a `EV_DENY` or `EV_GRANT` signal is received from the control program. If a grant signal is received, the ship enters the granted mode, where it stays until it is 250 meters before the bridge. If, on the other hand, the ship is denied passage, it enters a denied state, in which it stays until either a grant signal is received or the distance to the bridge is 250 meters. In the first case it enters the granted mode, in the latter it enters a waiting mode.

From the granted mode, the ship enters the passing mode when it is 250 meters from the bridge. If in waiting mode, the ship enters the passing mode only when a grant signal is received. When the passing mode is left, 100 meters after passing under the open bridge, the `EV_PASSED` signal is emitted and the passed mode is entered. Here it stays until between 2 and 3 kilometers away, yielding a cyclic behaviour of ships similar to that of the train automaton.

6.4 The Control Program

In this section we describe the control program used for operating the bridge. We shall first describe its design in terms of a state diagram, and then proceed to describe the implementation. In addition to this, we describe a number of mutations that we introduce into the control program. These mutated versions of the control program will be used for validation in section 6.5.

6.4.1 Design

The control program is built around the state diagram illustrated in figure 6.7. Transitions between states are triggered by events in the environment, values obtained by polling, and timers. At some transitions, a signal is sent to the environment in order to actuate according to the given situation.

Initially, the control program is in state closed. This denotes that the bridge is closed, but can be opened if a ship comes into range and sends a request. If, by polling, it is found that a train is approaching, the cleared signal is sent to the train and the control program switches to the train state, from which the bridge cannot be opened. It remains in this state until either the train has passed the bridge or a ship requests to pass under the bridge. In the first case, the control program returns to the closed state. In the latter, it sends a deny signal to the ship and switches to a state denoting that a ship is waiting for a grant.

From the closed and ship request states the bridge can be opened. As described in section 6.2.3, the first step of this procedure is to lower the barriers. This is done by sending a signal to the environment, after which the control program starts a timer that triggers after 10 seconds. When the timer triggers, the `EV_BRIDGE_RAISE` signal is sent to the bridge. When the environment emits the `EV_BRIDGE_UP` it can be assumed that the bridge is up, and the `EV_SHIP_GRANT` signal is given to the ship. When the `EV_SHIP_PASSED` signal is received, the bridge is closed and the barriers are lowered by a procedure analogue to that of opening.

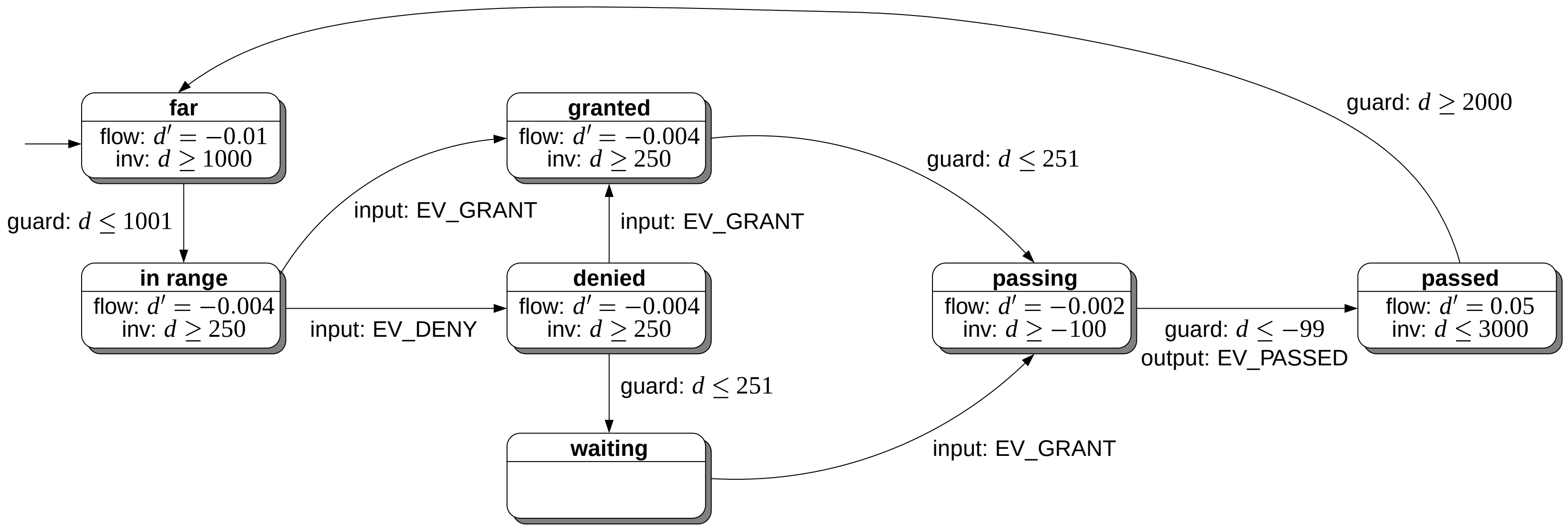

Figure 6.6: The ship Hybrid Automaton

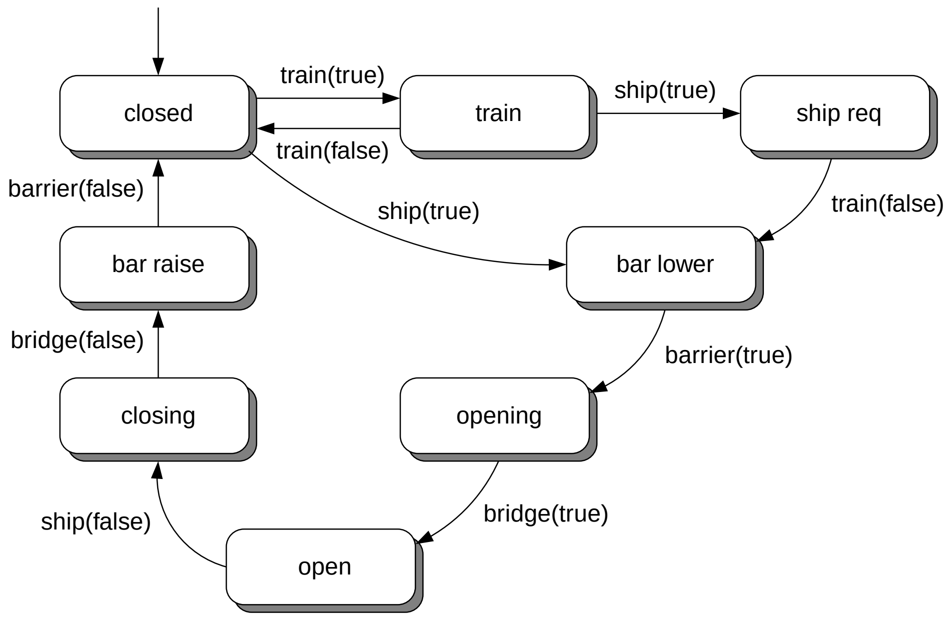

Figure 6.7: The state diagram of the control program

6.4.2 Implementation

The implementation of the control program is centred around the `CPState` class, partially listed in figure 6.8. It contains an integer field, `state`, that holds the current state and a boolean field, `train`, that is assigned the value true if a train is recognised while in the procedure of opening and closing the bridge. The latter variable allows the control program to clear an approaching train once the bridge has been closed. This functionality, however, is not described in the following.

The class has three methods which are invoked by active objects when a change in state has been observed. Their signatures are `public void train(boolean incoming)`, `public void bridge(boolean open)`, and `public void barrier(boolean up)`. Notice that only the method body of the first of these is shown in figure 6.8.

The operation performed by the active objects is to observe when actions occur in the environment. They then invoke the methods on the `CPState` instance in order to reflect the change in state. The pattern of method invocation is depicted in figure 6.8.

As an example of an active object, consider the class `TrainPoller` presented in figure 6.9. This class performs a periodic task with the purpose of recognising the events that a train reaches one of the sensoring zones on either side of the bridge. It invokes the `train(boolean incoming)` method in the instance of `CPState` with a `true` parameter value if a train is approaching and a `false` parameter value if the train has passed the bridge.

In the `run()` method of the `TrainPoller` class, the last polled value of the train's distance to the bridge is used to decide whether the train is approaching the bridge (incoming) or

```
import javax.realtime.*;
import environment.*;

public class CPState{
    /* define states as constants */
    public static final int CP_CLOSED = 0;
    public static final int CP_TRAIN = 1;
    public static final int CP_TRAIN_REQ = 2;
    public static final int CP_OPENING = 3;
    public static final int CP_OPEN = 4;
    public static final int CP_CLOSING = 5;
    public static final int CP_BAR_LOWERING = 6;
    public static final int CP_BAR_RAISING = 7;
    public static final int CP_BAR_DOWN = 8;
    public static final int CP_BAR_UP = 9;

    private static final int BAR_WAIT = 1100; // ms -- change at speedup
    private boolean train = false;
    private int state = CP_CLOSED;
    private Object lock = null;

    public void train(boolean incoming){
        this.train = incoming;
        RealtimeThread.currentRealtimeThread().beginSynchronization(lock);
        synchronized(lock){
            if(incoming){
                if(this.state == CP_CLOSED){
                    this.state = CP_TRAIN;
                    AbstractEnvironment.instance().actuate("EV_TRAIN_CLEARED");
                }
            }
            else{
                if(this.state == CP_TRAIN){this.state = CP_CLOSED;}
                if(this.state == CP_TRAIN_REQ){
                    this.state = CP_BAR_LOWERING;
                    AbstractEnvironment.instance().actuate("EV_BAR_LOWER");
                    /* start a timer that informs cp when barrier is down */
                    AsyncEventHandler aeh = new BarrierHandler(false, this);
                    OneShotTimer t = new OneShotTimer(new RelativeTime(BAR_WAIT),
                                                  aeh);
                    t.start();
                }
            }
        }
        RealtimeThread.currentRealtimeThread().endSynchronization(lock);
    }
...
}
```

Figure 6.8: Part of the CPState class maintaining control program state information

```
public class TrainPoller extends RealtimeThread{

    CPState state = null;
    private static final int POLL_PERIOD = 100; // poll every 100 ms
    private static final double DIST_APPROACH_FROM = 8000.0;
    private static final double DIST_APPROACH_TO = 2000.0;
    private static final double DIST_PASSED_FROM = -100.0;
    private static final double DIST_PASSED_TO = -200.0;

    private boolean train = false;
    private boolean incoming = true;
    private double last = -1;

    public TrainPoller(CPState state){
        super(new PriorityParameters(PriorityScheduler.instance().getNormPriority()),
              new PeriodicParameters(new AbsoluteTime(AbstractEnvironment.START_TIME),
                                     new RelativeTime(POLL_PERIOD), null, null, ...);
        this.state = state;
    }

    public void run(){
        awaitRelease();
        double dist;
        while(true){
            dist = AbstractEnvironment.instance().pollValue("Train", "distance");
            if(last != -1){
                if(last > dist){incoming = true;}
                else{incoming = false;}
            }
            last = dist;

            if(train){
                if((dist <= DIST_PASSED_FROM) && (dist >= DIST_PASSED_TO) && !incoming){
                    train = false;
                    state.train(false);
                }
            }
            else{
                if((dist <= DIST_APPROACH_FROM) && (dist >= DIST_APPROACH_TO) && incoming){
                    train = true;
                    state.train(true);
                }
            }
            waitForNextPeriod();
        }
    }
}
```

Figure 6.9: The class used for polling for polling for trains

not. This is necessary because the Hybrid Automaton modelling trains in the environment is cyclic (see section 6.3.3). As a consequence the distance to the bridge is within the interval twice per cycle – whereas the train is in fact only approaching once.

We now return our focus to the `CPState` class. When the `train(boolean incoming)` method is called with a parameter value of `true`, the train is cleared only if the bridge is closed. This is done by invoking the `actuate(String event)` method on the instantiated environment. The state is then changed to denote that fact that a train has been cleared.

If, on the other hand, the value of the incoming variable is `false`, the state is set to closed if no ship is waiting. Otherwise, the bridge opening procedure is initiated by sending a signal to the environment to lower the bars. Furthermore, a timer is started which, at the time of triggering, will inform the control program that the barriers are down in fashion similar to the event that a train is approaching.

Several threads invoke methods on the shared instance of `CPState`. This may lead to concurrent updates of the state in the control program. An example could be that the `TrainPoller` instance invokes the `train(boolean incoming)` at the time the handler for the event that bridge has been closed is executing. Thus, the methods for updating the current state are synchronised in order to obtain atomicity. Notice that we have included the invocations of the `beginSynchronization(Object o)` and `endSynchronization(Object o)` in the listing. As mentioned in section 3.1, these are required to obtain the behaviour described in the Real-Time Specification for Java.

6.4.3 Control Program Mutations

Based on the implementation described above, we have created four mutated control programs. These version have the potential to actuate on the environment in a manner leading to an undesirable situation. We first describe these situations as properties in terms of the mode constraints presented in section 4.5.

Properties

For the purpose of validating the four control programs described in section 6.4.3 a number of environment properties that are required to hold are needed. These include safety properties stating that not accidents occur in the environment. Examples of such properties are that no train ever crosses the bridge while it is not closed and that the bridge never starts to raise before the barriers are lowered. In addition, we introduce the requirement that no train should ever have to stop.

We state three such requirements that we have found to suffice for recognising that undesirable situations arise using the erroneous control programs. These are presented below:

- *Ship not granted when train cleared.* That is, if a train is cleared, no ship is ever granted. Stated in the mode constraint properties supported by the current implementation, this corresponds to: Train in mode cleared implies not Ship in mode granted.
- *Barriers down when bridge raising.* When the bridge is opening, the barriers must be down. Corresponds to: Bridge in mode raising implies Barriers in mode down.

- *Never train stopped.* The simple requirement that a train is never stopped translates to: true implies not Train in mode stopped.

Mutations

The four control programs which we shall use for validation contains one of the mutations:

- *Early raising of the bridge.* According to the procedure for opening the bridge, the barriers can be assumed to be lowered ten seconds after a signal has been sent to them. In this implementation, the control program waits for only eight seconds.
- *Missing detection of trains when bridge open.* When the bridge is open, a train cannot be cleared. However, a cleared signal must be sent once the bridge is closed. The error introduced here is that if a train is approaching while the bridge is up, it is never cleared.
- *Sending wrong signal to ships.* If a train is cleared, requests from ships should be denied. In this version, the control program may mistakenly send the wrong signal in this situation. That is, a ship is granted to pass under the bridge although it cannot be opened. The error is implemented so that, each time a ship is to be denied, there is a certain probability that the grant signal is sent. For the validation we have chosen the two probabilities 0.1 and 0.9.
- *Missing detection of train having passed the bridge.* The control program detects that a train has passed the bridge when it enters the polling zone after the bridge. By increasing the period of the polling thread, we introduce a risk that the train is not detected after passing the bridge.

6.5 Validation

Having defined our control program and the environment in which we wish it to control, we now continue to perform the validation of the mutated control programs. The properties stated in the previous section are used to try to detect errors.

6.5.1 Execution and Results

The four erroneous control programs were executed in parallel with the code generated from the Hybrid Automata. The environment was explored with the strategy currently implemented, that is, the approach where the environment stays in all modes as long as possible.

The properties listed in section 6.4.3 are implemented using the classes `Constraint` and the descendants of the `Property` interface. As an example, the constraint stating that the barriers must be down when the bridge is raising is listed in figure 6.10.

With the given modelling of the environment, it takes an unreasonable amount of time for the entities in the environment to approach the bridge. Therefore, the idea of speeding up

```
constraint = new Constraint(env,
                            "Barriers::down when Bridge::raising",
                            new ModeProperty("Bridge", "raising", false),
                            new ModeProperty("Barrier", "down", false));
env.addConstraint(constraint);
```

Figure 6.10: The constraint that barriers must be down when the bridge is raising stated in Java code

the simulation was appealing. A similar approach has been carried out for model checking in Uppaal ([HL02]).

A speed-up requires both the environment and the control program to execute at a faster pace, which requires adaption of the timing scopes as well as a fast enough test platform to allow the control program to finish its tasks within the new deadlines. The latter turned out not to constitute a problem, as the load of the system on which the tests were conducted was very low.

No support for performing speed-ups is provided by the execution framework – and it must therefore be done manually. In the environment, this is easily accomplished by multiplying all coefficients of the flows by some factor – in this case we chose 10. In the control program, only one of the tasks depends on time, namely the timer started to inform the control program when the barriers are either down or up. We therefore divide the time waited by this timer by ten in order to match the faster environment.

Since some of the errors introduced are not deterministic, the control programs were executed 8 times in order to observe the variation in validation durations. The results are presented in table 6.2.

In all executions, an error was found. In most cases, the errors are found within four minutes – except in the implementation where a wrong signal is possibly sent to a ship. However, this is to be expected due to the nondeterministic nature of this misbehaviour.

6.6 Conclusions on the Case Study

From the validation performed of the case study in the previous sections, we conclude that the approach of validating control programs in simulated environments is successful. We also conclude that the current implementation of the validation approach in the inquisitor framework works. However, a problem regarding polling the environment for the current value of some variable leads to problems when sporadic entities are modelled by cyclic automata.

We also conclude that speeding up the validation process is both desirable and possible. We therefore mention it in the future work presented in chapter 7.

Control Program	**Violated Property**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**	**Avg.**
Early raising of the bridge	Barriers down when bridge raising	67 s	68 s	67 s	68 s	68 s	68 s	66 s	68 s	67.5 s
Missing detection of trains when bridge open	Never train stopped	188 s	188 s	186 s	187 s	188 s	188 s	187 s	188 s	178.5 s
Sending wrong signal to ships (probability 0.1)	Not ship granted when train cleared	31 s	31 s	1703 s	1073 s	197 s	196 s	197 s	1169 s	574.6 s
Sending wrong signal to ships (probability 0.9)	Not ship granted when train cleared	198 s	199 s	31 s	31 s	31 s	31 s	31 s	31 s	72.9 s
Missing detection of train having passed the bridge	Never train stopped	188 s	188 s	186 s	187 s	188 s	188 s	187 s	188 s	178.5 s

Table 6.2: The results of 8 execution of each control program. The property mentioned is that broken by the given run.

Conclusion

CHAPTER 7

As computing hardware becomes smaller and more powerful, its usage for so-called *embedded systems* increases. Such systems are embedded into special-purpose devices and interact with other entities. Often, these entities make up an *environment* that must be *controlled* by the embedded system. That is, the embedded system acts as a *control program* which must keep its surrounding environment safe.

A special characteristic about such systems is that they often incorporate *real-time requirements*. In order to behave correctly, the interaction between the control program and its environment must occur at the right time. A typical example of such timing constraints is that the control program must *react* to some *stimuli* from the environment within a fixed interval of time.

Many of the systems described above are commonly used to control environments in which errors have fatal consequences. Others are shipped in large quantities, making errors hard and expensive to correct. Thus, a lot of effort is spent *validating* such systems for errors before they are put into use. Similarly, a large amount of research is conducted with the aim of improving the methods for validating embedded real-time systems.

In this project, we investigate a validation method based on executing the control program and observing its behaviour. Research in such techniques, known as *run-time validation*, has enjoyed increasing attention recently ([KKL$^+$01] and [HP00]). We have identified two shortcomings in current run-time validation techniques with regard to the validation of embedded real-time systems. One, support for real-time systems is relatively limited, and two, the control programs must be executed in their actual environments.

We propose a method that allows an embedded real-time system to be validated in a *simulated* environment. The environment is represented by a formal description of its possible behaviour. This approach is commonly used in model checking practices, where models of the control program interacts with models of the environment. The *model* of the environment incorporates the timing requirements imposed on the control program, thereby increasing the support for validation of control programs with real-time constraints. A further contribution is that the validation is performed in terms of the behaviour of the environment, rather than that of the control program, which allows a higher level of abstraction from the implementation.

Based on analysis of the generic behaviour and communication patterns of environments, we have found the notion of *Hybrid Automata* to be suitable for formally representations of environments. We have, however, introduced a number of modifications to the general theory, in order to support simulation and interaction with control programs.

We have presented an approach to performing environment simulation based on formal models of the allowed environment behaviour. Using this approach, generated code simulating the environment is executed in parallel with the control program. In order to obtain an interesting *coverage* of the possible environment behaviour, we have considered several *strategies* to exploring the environment models.

Furthermore, we have suggested an approach by which requirements on the behaviour of the control program can be stated in terms of simple constraints in the behaviour of the environment. Specifically, the specification of real-time constraints is supported by modelling in the environment. Thereby, errors in the control program (whether timing-related or not) can be found by observing the state of the environment during runtime.

In order to investigate the feasibility of the proposed method, we have implemented a prototype and carried out a case study. This case study exemplifies the process of formalisation and validation. The results of the validation performed show encouraging results.

7.1 Future Work

In this section, we present a number of possible future enhancements to the work presented in this report. Some are directly related to improving the applicability of the method, others show how the method can be applied in a greater perspective.

- *Applicability-related extensions.* In this project, we have not considered the dynamic creations of entities in the environment. Since such creations in fact do occur in environments, a more intuitive modelling can be accomplished by supporting them. It is suggested that the dynamic creation is mapped on to a sporadic update of the environment. Thereby, dynamic entities can be created by a jump from one mode to another.

  Another way to increase the applicability of the method is to implement generic support for speeding up the validation process. A manual way to increase the speed of the validation was used in the case. In a more generic setting, the speed-up could be supported by the Inquisitor Framework. For example, by speeding up the progression of logical time by some factor, all timing scopes in the control program and environment could be easily changed by the given factor.

  The last applicability improvement we shall consider is to visualise the interaction between the control program and the environment during simulation. Thereby, the testing team would have an easier job identifying the flaw in the control program.

- *Validating external control programs.* The prototype presented in this report supports the validation of real-time systems in a setting where logical time can be stopped. Thus, the time spent by the simulator for updating the environment is not limited by the length of the simulation period.

  In a more realistic setting, the control program is executed on specialised hardware. By implementing a stub for interaction with the simulated environment, such control programs could be validated even though they are executed on a separate processing unit. In order to accomplish this, however, a limit on the time spent for updating the environment during simulation must be adhered to.

◦ *Control program specifications.* By assuming the presence of a specification of the control program, a couple of enhancements to the validation method can be implemented. First, the knowledge about the control program can be used to obtain a greater coverage of its possible behaviour. The knowledge may also be used to identify critical states in the control program. Such knowledge may in turn be used to decide the simulation of the environment.

  A second use of the control program specification is to observe whether the implementation behaves in accordance with the specification. For example, the conformity criterion used for model-based testing could be used. The extended knowledge about the control program may be useful for better debugging support. For example, when an error in the environment occurs, the actions of the control program can be deduced from the model. If a conformity error occurred before the error in the environment, it is likely that the implementation does not conform to the specification. Otherwise, the specification of the control program may in fact allow the given behaviour.

  A third use of the control program specification is to perform a model-check of the control program alongside the environment models. For example, if the control program specification was given as a Hybrid Automata the HyTech model checker could be used. In order to allow model checking, however, the usual restrictions on the environment automata must be applied.

Source Code Listings APPENDIX A

This appendix includes the source code listings referred in the report.

A.1 Document Type Definition

The environment specification must be described in XML files adhering to the following format:

```
<!ELEMENT environment (event*, automaton+, instantiation+)>
<!ATTLIST environment name CDATA #REQUIRED>
<!ELEMENT event EMPTY>
<!ATTLIST event label CDATA #REQUIRED
                type CDATA #REQUIRED>
<!ELEMENT automaton (var*, param*, mode+, transition*)>
<!ATTLIST automaton name CDATA #REQUIRED
    dynamic CDATA #REQUIRED>
<!ELEMENT var EMPTY>
<!ATTLIST var name CDATA #REQUIRED
              defaultval CDATA #REQUIRED
              parameter CDATA #REQUIRED>
<!ELEMENT mode (update?, invariant*)>
<!ATTLIST mode name CDATA #REQUIRED
               initial CDATA #REQUIRED>
<!ELEMENT transition (source, target, guard*, weight, input?, output?, update?)>
<!ELEMENT source EMPTY>
<!ATTLIST source mode CDATA #REQUIRED>
<!ELEMENT target EMPTY>
<!ATTLIST target mode CDATA #REQUIRED>
<!ELEMENT update (#PCDATA)>
<!ELEMENT invariant (#PCDATA)>
<!ELEMENT guard (#PCDATA)>
<!ELEMENT weight (#PCDATA)>
<!ELEMENT input (#PCDATA)>
<!ELEMENT output (#PCDATA)>
<!ELEMENT instantiation (#PCDATA)>
```

A.2 Partial Barrel Environment Source

This section lists the source code generated from the partial specification of the barrel environment in section 5.3.1.

A.2.1 The Barrel Environment Class

The `BarrelEnvironment` class is defined as follows:

```
// package environment.generated;

import environment.*;

public class BarrelEnvironment extends AbstractEnvironment {

    public BarrelEnvironment() {
        super("BarrelEnvironment");
        addEvent("EV_START", Event.INPUT);
        addEvent("EV_STOP", Event.INPUT);
        addEvent("EV_ERR", Event.OUTPUT);
        addEvent("EV_OK", Event.OUTPUT);

        addAutomata(new Barrel(50));
    }
}
```

A.2.2 The Barrel Class

The `Barrel` class is defined as follows:

```
// package environment.generated;

import environment.*;

public class Barrel extends AbstractAutomata {

    public double h = 0;

    public Barrel(int h) {
        super("Barrel", false);

        /* instance initialization */
        this.h = h;

        /* adding modes */
        addMode(new AbstractMode(this, "filling") {

                    public void update(AbstractAutomata me, int millis) {
                        ((Barrel) me).height = ((Barrel) me).height + millis * 0.001;
                    }
                }
                , true);

        ((AbstractMode)modes.get("filling"))
        .addInvariant(new AbstractEvaluation(this) {
                          public boolean evaluate(AbstractAutomata me) {
                              return ((Barrel) me).height <= 100;
                          }
                      }
                     );


        addMode(new AbstractMode(this, "pumping") {

                    public void update(AbstractAutomata me, int millis) {
                        ((Barrel) me).height = ((Barrel) me).height - millis * 0.005;
                    }
                }
                , false);

        ((AbstractMode)modes.get("pumping"))
        .addInvariant(new AbstractEvaluation(this) {
```

```
                  public boolean evaluate(AbstractAutomata me) {
                      return ((Barrel) me).height > 9.9;
                  }
              }
             );


addMode(new AbstractMode(this, "overflow") {

            public void update(AbstractAutomata me, int millis) {
            }
        }
        , false);

((AbstractMode)modes.get("overflow"))
.addInvariant(new AbstractEvaluation(this) {
                  public boolean evaluate(AbstractAutomata me) {
                      return true;
                  }
              }
             );

/* adding transitions */
addTransition(new AbstractTransition("transition_0", ((AbstractMode)modes.get("filling"))
                                     , ((AbstractMode)modes.get("pumping"))
                                    ) {
                  public void update() {
                  }
              }
             );

((AbstractTransition)transitions.get("transition_0"))
.addGuard(new AbstractEvaluation(this) {
              public boolean evaluate(AbstractAutomata me) {
                  return ((Barrel) me).height >=90;
              }
          }
         );


((AbstractTransition)transitions.get("transition_0"))
.setInputEvent(AbstractEnvironment.instance().getEventByLabel("EV_START"));


((AbstractTransition)transitions.get("transition_0"))
.setWeight(1);

addTransition(new AbstractTransition("transition_1", ((AbstractMode)modes.get("pumping"))
                                     , ((AbstractMode)modes.get("filling"))
                                    ) {
                  public void update() {
                  }
              }
             );

((AbstractTransition)transitions.get("transition_1"))
.addGuard(new AbstractEvaluation(this) {
              public boolean evaluate(AbstractAutomata me) {
                  return ((Barrel) me).height < 20;
              }
          }
         );
        ((AbstractTransition)transitions.get("transition_1"))
        .setInputEvent(AbstractEnvironment.instance().getEventByLabel("EV_STOP"));

        ((AbstractTransition)transitions.get("transition_1"))
        .setWeight(1);

        addTransition(new AbstractTransition("transition_2", ((AbstractMode)modes.get("filling"))
                                             , ((AbstractMode)modes.get("overflow"))
                                            ) {
                          public void update() {
                          }
                      }
                     );

        ((AbstractTransition)transitions.get("transition_2"))
        .addGuard(new AbstractEvaluation(this) {
                      public boolean evaluate(AbstractAutomata me) {
                          return ((Barrel) me).height >= 99.9;
                      }
                  }
                 );

        ((AbstractTransition)transitions.get("transition_2"))
        .setWeight(1);
    }
}
```

A.3 The Bridge Environment Specification

The specification of the bridge environment is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE environment SYSTEM "eha.dtd">
<environment name="BridgeEnvironment">
  <event label="EV_BRIDGE_RAISE" type="INPUT"/>
  <event label="EV_BRIDGE_LOWER" type="INPUT"/>
  <event label="EV_TRAIN_CLEARED" type="INPUT"/>
  <event label="EV_SHIP_DENY" type="INPUT"/>
  <event label="EV_SHIP_GRANT" type="INPUT"/>
  <event label="EV_SHIP_REQUEST" type="OUTPUT"/>
  <event label="EV_SHIP_PASSED" type="OUTPUT"/>
  <event label="EV_BRIDGE_OPEN" type="OUTPUT"/>
  <event label="EV_BRIDGE_CLOSED" type="OUTPUT"/>
  <event label="EV_BAR_RAISE" type="INPUT"/>
  <event label="EV_BAR_LOWER" type="INPUT"/>
  <event label="EV_RAIL_DETACH" type="INPUT"/>
  <event label="EV_RAIL_ATTACH" type="INPUT"/>
  <automaton name="Bridge" dynamic="false">
    <var name="degree" defaultval="0" parameter="false"/>
    <mode name="down" initial="true">
      <invariant>true</invariant>
    </mode>
    <mode name="raising" initial="false">
      <update>me.degree = me.degree + millis * 0.05;</update>
      <invariant>me.degree &lt;= 80</invariant>
    </mode>
    <mode name="up" initial="false">
      <invariant>true</invariant>
    </mode>
    <mode name="lowering" initial="false">
      <update>me.degree = me.degree - millis * 0.05;</update>
      <invariant>me.degree &gt;= 0</invariant>
    </mode>
    <transition>
      <source mode="down"/>
      <target mode="raising"/>
      <guard>true</guard>
      <weight>1</weight>
      <input>EV_BRIDGE_RAISE</input>
    </transition>
    <transition>
      <source mode="raising"/>
      <target mode="up"/>
      <guard>me.degree &gt;= 79 </guard>
      <weight>1</weight>
      <output>EV_BRIDGE_OPEN</output>
    </transition>
    <transition>
      <source mode="up"/>
      <target mode="lowering"/>
      <guard>true</guard>
      <weight>1</weight>
      <input>EV_BRIDGE_LOWER</input>
    </transition>
    <transition>
      <source mode="lowering"/>
      <target mode="down"/>
      <guard>me.degree &gt;= 0 </guard>
      <weight>1</weight>
      <output>EV_BRIDGE_CLOSED</output>
    </transition>
  </automaton>
  <automaton name="Train" dynamic="false">
    <var name="distance" defaultval="10000" parameter="true"/>
    <mode name="far" initial="true">
      <update>me.distance = me.distance - millis * 0.2;</update>
      <invariant>me.distance &gt;= 8000</invariant>
    </mode>
    <mode name="approaching" initial="false">
      <update>me.distance = me.distance - millis * 0.2;</update>
      <invariant>me.distance &gt;= 500</invariant>
    </mode>
    <mode name="stopped" initial="false">
      <invariant>true</invariant>
    </mode>
    <mode name="cleared" initial="false">
      <update>me.distance = me.distance - millis * 0.15;</update>
      <invariant>me.distance &gt;=-200</invariant>
    </mode>
    <mode name="passed" initial="false">
      <update>me.distance = me.distance + millis * 0.2;</update>
      <invariant>me.distance &lt;=12000</invariant>
    </mode>
    <transition>
      <source mode="far"/>
      <target mode="approaching"/>
      <guard>me.distance &gt;= 7999</guard>
      <weight>1</weight>
    </transition>
    <transition>
      <source mode="approaching"/>
      <target mode="stopped"/>
      <guard>me.distance &lt;= 501</guard>
      <weight>1</weight>
    </transition>
```

```
    <transition>
      <source mode="approaching"/>
      <target mode="cleared"/>
      <guard>me.distance &gt;= 1500</guard>
      <weight>1</weight>
      <input>EV_TRAIN_CLEARED</input>
    </transition>
    <transition>
      <source mode="stopped"/>
      <target mode="cleared"/>
      <guard>true</guard>
      <weight>1</weight>
      <input>EV_TRAIN_CLEARED</input>
    </transition>
    <transition>
      <source mode="cleared"/>
      <target mode="passed"/>
      <guard>me.distance &lt;= -199</guard>
      <weight>1</weight>
    </transition>
    <transition>
      <source mode="passed"/>
      <target mode="far"/>
      <guard>me.distance &gt;= 10000</guard>
      <weight>1</weight>
    </transition>
  </automaton>
  <automaton name="Ship" dynamic="false">
    <var name="distance" defaultval="2000" parameter="true"/>
    <mode name="far" initial="true">
      <update>me.distance = me.distance - millis * 0.1;</update>
      <invariant>me.distance &gt;=999</invariant>
    </mode>
    <mode name="inrange" initial="false">
      <update>me.distance = me.distance - millis * 0.04;</update>
      <invariant>(me.distance &gt;=250)</invariant>
    </mode>
    <mode name="granted" initial="false">
      <update>me.distance = me.distance - millis * 0.04;</update>
      <invariant>me.distance &gt;=250</invariant>
    </mode>
    <mode name = "denied" initial="false">
      <update>me.distance = me.distance - millis * 0.04;</update>
      <invariant>me.distance &gt;=250</invariant>
    </mode>
    <mode name="waiting" initial="false">
      <invariant>true</invariant>
    </mode>
    <mode name="passing" initial="false">
      <update>me.distance = me.distance - millis * 0.02;</update>
      <invariant>me.distance &gt;= -100</invariant>
    </mode>
    <mode name="passed" initial="false">
      <update>me.distance = me.distance + millis * 0.5;</update>
      <invariant>me.distance &lt;=3000</invariant>
    </mode>
    <transition>
      <source mode="far"/>
      <target mode="inrange"/>
      <guard>me.distance &lt;=1000</guard>
      <weight>1</weight>
      <output>EV_SHIP_REQUEST</output>
    </transition>
    <transition>
```

```
      <source mode="inrange"/>
      <target mode="granted"/>
      <guard>true</guard>
      <weight>1</weight>
      <input>EV_SHIP_GRANT</input>
    </transition>
    <transition>
      <source mode="inrange"/>
      <target mode="denied"/>
      <guard>true</guard>
      <weight>1</weight>
      <input>EV_SHIP_DENY</input>
    </transition>
    <transition>
      <source mode="denied"/>
      <target mode="waiting"/>
      <guard>me.distance &lt;= 251</guard>
      <weight>1</weight>
    </transition>
    <transition>
      <source mode="granted"/>
      <target mode="passing"/>
      <guard>me.distance &lt;= 251</guard>
      <weight>1</weight>
    </transition>
    <transition>
      <source mode="waiting"/>
      <target mode="passing"/>
      <guard>true</guard>
      <weight>1</weight>
      <input>EV_SHIP_GRANT</input>
    </transition>
    <transition>
      <source mode="passing"/>
      <target mode="passed"/>
      <guard>me.distance &lt;= -99</guard>
      <weight>1</weight>
      <output>EV_SHIP_PASSED</output>
    </transition>
    <transition>
      <source mode="passed"/>
      <target mode="far"/>
      <guard>me.distance &gt;= 2000</guard>
      <weight>1</weight>
    </transition>
  </automaton>
  <automaton name="Barrier" dynamic="false">
    <var name="degree" defaultval="0" parameter="false"/>
    <mode name="up" initial="true">
      <invariant>true</invariant>
    </mode>
    <mode name="lowering" initial="false">
      <update>me.degree = me.degree + millis * 0.1;</update>
      <invariant>me.degree &lt;= 90</invariant>
    </mode>
    <mode name="down" initial="false">
      <invariant>true</invariant>
    </mode>
    <mode name="raising" initial="false">
      <update>me.degree = me.degree - millis * 0.1;</update>
      <invariant>me.degree &gt;= 0</invariant>
    </mode>
    <transition>
      <source mode="up"/>
      <target mode="lowering"/>
      <guard>true</guard>
      <weight>1</weight>
      <input>EV_BAR_LOWER</input>
    </transition>
    <transition>
      <source mode="lowering"/>
      <target mode="down"/>
      <guard>me.degree &gt;= 89 </guard>
      <weight>1</weight>
    </transition>
    <transition>
      <source mode="down"/>
      <target mode="raising"/>
      <guard>true</guard>
      <weight>1</weight>
      <input>EV_BAR_RAISE</input>
    </transition>
    <transition>
      <source mode="raising"/>
      <target mode="up"/>
      <guard>me.degree &lt;= 1 </guard>
      <weight>1</weight>
    </transition>
  </automaton>
  <automaton name="Rails" dynamic="false">
    <var name="distance" defaultval="0" parameter="false"/>
    <mode name="attached" initial="true">
      <invariant>true</invariant>
    </mode>
    <mode name="detaching" initial="false">
      <update>me.distance = me.distance + millis * 0.1;</update>
      <invariant>me.distance &lt;= 20</invariant>
    </mode>
    <mode name="detached" initial="false">
      <invariant>true</invariant>
    </mode>
    <mode name="attaching" initial="false">
      <update>me.distance = me.distance - millis * 0.1;</update>
      <invariant>me.distance &gt;= 0</invariant>
    </mode>
    <transition>
      <source mode="attached"/>
      <target mode="detaching"/>
      <guard>true</guard>
      <weight>1</weight>
      <input>EV_RAIL_DETACH</input>
    </transition>
    <transition>
      <source mode="detaching"/>
      <target mode="detached"/>
      <guard>me.distance &gt;= 19.9 </guard>
      <weight>1</weight>
    </transition>
    <transition>
      <source mode="detached"/>
      <target mode="attaching"/>
      <guard>true</guard>
      <weight>1</weight>
      <input>EV_RAIL_ATTACH</input>
    </transition>
    <transition>
      <source mode="attaching"/>
      <target mode="attached"/>
      <guard>me.distance &lt;= 0.1 </guard>
      <weight>1</weight>
    </transition>
  </automaton>
  <instantiation>Bridge()</instantiation>
  <instantiation>Train(10000)</instantiation>
  <instantiation>Ship(2000)</instantiation>
  <instantiation>Barrier()</instantiation>
  <instantiation>Rails()</instantiation>
</environment>
```

Bibliography

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[BW01] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 2001.

[CDH$^+$00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[Chr02] Anders B. Christensen. *The Inquisitor Framework – A Framework for Validation of Real-Time Systems*. Aalborg University, Technical Report, 2002.

[CSL$^+$87] D. Cornhill, L. Sha, L. Lehoczky, J. Rajkumar, and H. Tokuda. *Limitations of Ada real-time scheduling*. Proceedings of the International Workshop on Real Time Ada Issues, ACM Ada Letters, 1987.

[DH99] Matthew B. Dwyer and John Hatcliff. Slicing software for model construction. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 105–118, 1999.

[DHJ$^+$00] M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, R. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification, 2000.

[dVT00] René G. de Vries and Jan Tretmans. On-the-fly conformance testing using spin. *Software Tools for Technology Transfer*, 2(4):382–393, March 2000.

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):110–122, 1997.

[HL02] Martijn Hendriks and Kim G. Larsen. Exact acceleration of real-time model checking. 2002.

[Hol97] Gerard J. Holzmann. The spin model checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[Hol00] G. J. Holzmann. Software model checking. Marktoberdorf, Germany, 2000.

[HP00] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4), April 2000.

[JCTG96] J. C. Fernandez, C. Jard, T. Jeron, and G. Viho. Using on-the-fly verification techniques for the generation of test suites. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 348–359, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[KKL$^+$01] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Java-mac: a run-time assurance tool for java programs. In Klaus Havelund and Grigore Rosu, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[RTJ02] The Real-Time Java Export Group RTJEG. *The Real-Time Specification for Java*. Addison Wesley, 2002.

[Yov97] Sergio Yovine. Kronos: A verification tool for real-time systems. *Journal of Software Tools for Technology Transfer*, 1(1–2), Oct 1997.