Dynamic Analysis Overview and a Proposed Verification Tool for Temporal Properties in Security-Critical Software

Christian Colombo

February 25, 2008

Abstract

The need for correct software is increasing as computers are proliferating in every aspect of our lives. Dynamic analysis is a possible way of increasing the reliability of software by introducing a monitoring and verification mechanism over and above a computer system, so that if under some unprecedented circumstance, any of its specifications are violated, an alarm will be raised. This paper gives an overview of the literature in the subject and also puts forward a proposal of further research and investigation which seems to be very promising.

1 Introduction

As computer systems become increasingly present in all the aspects of our lives, be it in avionics, or any other form of transport, be it medical equipment, be it an on-line billing system, it becomes increasingly important to provide reliable and robust software. Faults in security-critical systems can at best cost a lot of money and at worse cause the loss of human lives.
Thus far, the main approach to this problem was testing. Despite the effectiveness of proper testing, it is extremely difficult to test huge software products in a sufficiently thorough manner so as to ensure their correctness. The reason is that testing lacks coverage. One must bear in mind that a system does not work in a vacuum, but rather in an environment which is not in our control. Usually, we cannot predict (and much less simulate) all the environment behaviours to test our system in every possible situation. Therefore, another approach to provide secure software is model checking. In this case, we try to verify all execution traces which the system can possibly run into. However, this is usually impractical on a system which is large enough to be of any practical use.
Therefore, it seems that we need to find another way of ensuring that computer software is reliable and robust. This is a sort of trade-off between testing and model-checking. We will take the scalability of testing and the reliability of model-checking by verifying an execution trace during the actual runtime of a

system. The idea is that we have a clear description of the system's specification representing all the acceptable behaviour and while the system is executing we are continually ensuring that all the behaviour is adhering to the specification. In this way we would be guaranteeing that whatever the environment or the input, the behaviour is still correct. In case a property violation is encountered, the verifying system can either raise an alarm or else, even more appropriately, make some action which corrects the state of the monitored system.
In order to provide a description of the specification, we need to have succinct and clear manner which is not error prone itself. A number of formal notations have been proposed, each of which were designed with a particular domain in mind. Such a notation can still be difficult for a developer without a background in formal notation, so we need to find easier and clearer ways of specifying the security properties.
However, there still remains the problem of inserting the monitoring code which takes note of the events occurring in the security-critical system and then verifying them against the specification. We need to keep the process of injecting this code also error prone and hence we need to find ways of automating it.
Finally, we need to consider the problem that upon injecting monitoring code, we have changed the environment of the system, because we have introduced an overhead to the actual system. This issue requires careful consideration because we do not want to introduce errors while trying to eliminate them. This applies more specifically to real-time systems because these are more error prone due to their strict timings.
In this paper we will give a literature review of the current trends in runtime verification. In Section 2 we will go into dynamic analysis, explaining its phases in Section 2.1 and its flavours in Section 2.2. We will then give a comparison of static and dynamic analysis in Section 2.3. Later on, we will give an account of possible notations (Section 2.4), instrumentation approaches (Section 2.5) and verification algorithms (Section 2.6). In Section 3, we start by a short introduction to synchronous languages (Section 3.1) which form an important part of our proposal. Then we give an account of the proposed directions for the research (Section 3.3) together with a brief description of what has been done so far (Section 3.4 and 3.5). Finally, in Section 4 we present our conclusions.

2 Dynamic Analysis

2.1 What is Dynamic Analysis?

Dynamic analysis is the process of verifying a system (referred to as the target system) while it is executing. In more technical terms, dynamic analysis will refer to all the architectures which analyse a particular execution path of the system to detect a possible violation of the specified system properties [Ern03]. To perform such an analysis, we would require two main very basic components over and above the target system: (i) a monitoring mechanism; (ii) a verification mechanism. Monitoring the system entails the elicitation of events while they are occurring. These events are then communicated to the verification mechanism which verifies that the sequence of events adheres to the target system's specification. If a violation is found, then, the verification system raises an alarm or can possibly react in some way so as to revert the target system back to an acceptable state. The following subsections will give a more detailed account of the phases which dynamic analysis comprises.

2.1.1 The Phases of Dynamic Analysis

We will now give an outline of the various phases involved in different dynamic analysis approaches. Subsequently, in the next section we will classify the different approaches according to the way these phases are tackled.

1. Specification – The first phase includes the specification of the system's properties in some kind of formal notation. This notation may be either a particular logic or automata depending on the domain of the problem.

2. Instrumentation – Once the system properties are specified, these must be instrumented into the system code. Usually this process also involves the adaptation of the specified property so that it can be checked on a single execution path of the system. Furthermore, the translation also entails the conversion of the specified properties in a format which is the same as the target system. Sometimes this translation also takes the form of generating monitoring code which concretely ensures that the specified properties are not violated. The extent of the translation process depends on the divergence between the properties' specification format and target system's format. As soon as the system properties are in the same form as the system itself, they can then be instrumented into the target code. In some cases the monitoring code is not instrumented in the system code. Rather, a monitoring system is run in parallel to the monitored system and this will circumvent the need of the full instrumentation of monitoring code (but just the code to generate the events).

3. Monitoring – Subsequently, the actual monitoring of the system is carried out. This can either be done on-line or off-line. This means that either the system is actually running and the monitoring mechanism is running in parallel or else a trace of the system's execution is saved and then it is verified at a later time.

4. Handling violation of properties – The next phase would be raise exceptions on the detection of a property violation. Another possibility is that rather than simply raise an exception (as a notify of the violation), an automatic fault-handling mechanism may be implemented. This means that part of system's functionality is actually implemented (by design) as the response to the fulfilment (or violation) of a certain condition.

5. Mitigating the impact of verification – A final possible phase is the mitigation phase. The purpose of this phase is to try to minimise or possibly

eliminate the impact of the instrumented monitoring code in the actual system. It is not always possible to simply remove the "extra" code. In the first place, considering the removal of the monitors assumes that the monitoring code is simply there for testing purposes, when in fact this is not always the case (as in the case where part of the system's functionality is implemented as "error-handling code"). Secondly, removing monitors from a system may bring about undesired effects which did not emerge due to the monitors. Two possible approaches to this problem would be to give guarantees (using metrics) on the effects of the added code, or/and allow the system to adapt itself (using reflection) at runtime to leave out certain code which checks for properties which may have been verified earlier in the execution.

2.2 Flavours of Dynamic Analysis

The following are various software design methodologies or architecture patterns which incorporate dynamic analysis to assure system properties. In each case we will explain which of the above phases the particular methodology includes. Furthermore, we will also comment on each variation as regards to the applicability to our research project.

2.2.1 Design by Contract

Design by contract [Mey92] has been developed with aim of creating more secure software by manually inserting checks in the code which correspond to a *contract*. These checks, better known as assertions are then verified during runtime. This approach is quite primitive when considering the amount of effort which is left to the developer to handle. In design by contract, the system properties are inserted as preconditions or post-conditions in methods or as invariants for classes. Hence there is no need for code instrumentation as such [CDR04], since the property specifications are very closely coupled to the actual system implementation. It is important to note that there is no option of off-line monitoring since the monitors become an integral part of the actual system. There is, however, the possibility of compiling a version of the program which simply ignores all the monitoring [Mey92].
In the verification of security-critical systems, the aim is that the verification process is automated as much as possible so that the possibility of human error is eliminated. If the error checking mechanism is itself error-prone, then it is not very useful. Therefore, this approach has quite a considerable disadvantage.

2.2.2 Runtime Verification

Runtime verification [CM05, ZKTR07, LKK$^+$99, BGHS04] has also been developed to verify software against a formal specification. However, it differs from design by contract in various aspects. First of all, the specification of properties is done through formal notation and it is usually automatically instrumented

into the target system. Runtime verification not only ensures that no system properties are violated, but it also provides the mechanism to correct the behaviour so that the system can continue to function. Therefore, the extra code for runtime verification includes both the monitoring code (which extracts the sequence of events (trace) from the current program execution) and the code which is triggered upon a violation of the properties.
If the specification of security properties is as cumbersome as writing the program itself, there is probably the same chance of making the same mistakes. Hence runtime verification is much more appropriate for security-critical systems because it uses formal notation for specifying security properties. The advantage is that usually, formal notation is very succinct and much more abstract than the actual implementation.
Another attractive aspect of runtime verification is that it allows the user to specify extra code so that the system which finds itself in a bad state, can be reverted back to a valid state. This is very desirable because it eliminates the need of human intervention upon a security violation.
Although strictly speaking runtime verification should occur during the runtime of the target system, variations of runtime verification allow for both on-line and off-line verification of the execution trace. This is motivated by the fact that the overhead introduced by the runtime verification code is sometimes too large. The alternative is limit the overhead to the extraction of the trace from the running program (similar to creating a log) and subsequently the actual verification of the trace is performed later on. This is further discussed in Section 2.2.5.

2.2.3 Monitoring-Oriented Programming

Monitoring-oriented programming is a paradigm which combines the specification and the implementation of a system [CR03]. It goes further than runtime verification in that it not only specifies properties to detect violations and raise exceptions, but the violation handling mechanism is itself part of the design of the system's functionality. Hence, the monitoring is not simply an extra check on top of the system but an integral part of the system's design. For example, Chen et al. give an example of a system which does user authentication using a security policy in monitoring-oriented programming [CDR04]. Monitoring-oriented programming is a very flexible architecture and allows for the monitoring code to be separated from the actual system's code [CDR04]. It also allows the monitoring code to be asynchronous with respect to the system being monitored [CDR04].

2.2.4 Runtime Reflection

Runtime reflection [LS07] adds yet another idea over and above monitoring-oriented programming methodology: it incorporates a diagnosis layer just after the monitoring layer. The purpose of this is layer is to identify the type of failure that occurred rather than simply detect that a failure has occurred. This enables the system to give an "explanation of the current system state"

[LS07]. One advantage of this extra layer is that it is more easily applicable in a distributed system where distributed parts send their monitoring information but the diagnosis is carried out at a central system. Furthermore, there is more separation of concerns in that the monitoring is simply concerned with obtaining values representing the state of the system, while it is up to the diagnosis layer to interpret that system state. Eventually, the mitigation (subsequent) layer performs an operation in response to that interpretation. It is important to note that the runtime reflection architecture pattern can still be achieved by using an monitoring-oriented programming methodology by implementing the diagnosis layer in the monitor-triggered code.
The idea of having an "explanation" for ending in a bad state is very desirable. This is more especially so when dynamic analysis is used during the testing phase to identify errors, since this would be of great help to the developers.

2.2.5 On-line vs. Off-line Verification

Recall that dynamic analysis is used verify the system's properties for one particular trace of events during the actual execution of the system. The monitors are responsible to record a trace of events which are eventually analysed. The analysis of an execution trace can be done on-line or off-line [CDR04]. On-line verification is when the target system and the verifying system are run synchronously in parallel (either as two separate systems or as a single instrumented system). The advantage of such a configuration is that the verification system can correct the detected property violation during the execution of the system. However, this poses more challenges for the verification process since the trace is not available as a whole, but becomes available progressively in synch with the actual execution. The fact that the trace is not all available means that certain very efficient algorithms which are able to verify the trace backwards [RH01, RH05] cannot be used. However, there still exist efficient algorithms which are able to work on-the-fly [HR02, HR04]. The main techniques used for runtime verification are dynamic programming [HR02] and rewriting [CELM96, HR01c, HR01b, HR04]
Choosing between on-line and off-line verification, one should consider whether the main purpose of the runtime verification is to find errors (E.g.: during testing) or to find and automatically correct errors. In the first case, it is practically irrelevant whether the verification is performed on-the-fly (on-line) or not (off-line). Hence, it is preferable to use off-line verification in such a circumstance. At the same time, off-line verification reduces the overhead of verification (since this is postponed to a later time and need not be done while the actual system is running). It is important to note that we still need the monitors to report the relevant events which are taking place.

2.3 Static vs. Dynamic Analysis

An important difference in software analysis is whether this is done before or during runtime. Static analysis will be used to refer to all the techniques (including

theorem provers and model checkers) used to verify a program for all possible execution paths (before the actual execution). On the other hand, dynamic analysis will include all the techniques used to verify the system's properties for one particular execution trace obtained by executing the program. Being able to verify properties for all possible executions paths [UT02, ZKTR07], makes static analysis a very desirable objective. However, for the algorithm to be tractable when applied to systems of a practical magnitude, static analysis depends on having a decidable domain. Various abstraction and reduction techniques have been proposed to scale up static analysis, but full verification of large-scale software systems is still largely unattainable [GH05, ZKTR07, FS04]. In contrast with static analysis, using dynamic analysis, one checks that a given system property holds along a particular execution path [ZKTR07]. This is particularly useful to ensure that at no time during the execution of the system, are any of the system properties violated. Conversely, it can also identify execution paths along which the properties to be verified are not satisfied [ZKTR07]. Essentially, dynamic analysis links the abstract specification to the actual concrete implementation [LBAK^{+}98, STY03]. Thus, dynamic analysis can be used as a protection from potential faults at runtime, by implementing monitors to react to any property violations encountered [GH05]. Another motivation for using dynamic analysis is that certain information is only available at runtime. Furthermore, behaviours of the system may possibly depend on the environment where the system is running [CM05].
Although, there is this fundamental difference between static analysis and dynamic analysis, ways have been proposed in which these two approaches can complement each other by exploiting the benefits of one to aid the other [Ern03, ZKTR07, CF00, AN07]. Ernst, in fact, argues that the difference between static analysis and dynamic analysis is over-emphasized [Ern03]. The complementarity can thus be achieved by applying both approaches and taking advantage of the "soundness" of static analysis while also benefiting of the "efficiency and precision" of dynamic analysis [Ern03]. The challenge described in [ZKTR07] (while implementing a similar approach to the one in [Ern03]) is that of using the same specification language for both the static analysis and dynamic analysis. Interestingly, the purpose of incorporating a runtime checker on top of the existing program (static) verification system (Jahob verification system) in [ZKTR07], was to help the developers in identifying errors in the system by showing concrete executions where the errors arise. In [CF00], static analysis is used to avoid unnecessary runtime checking.
Another way in which static analysis and dynamic analysis can complement each other is for test-case generation. Static analysis can be used to "intelligently" generate test cases for a dynamic analysis tool to find errors during testing [ABG^{+}05].

2.4 Logics and Automata for Dynamic Analysis

In this section we will consider various languages and logics that have been used for the specification of system properties. However, before going into the actual notations we need to provide a classification for these logics.

2.4.1 Models of Time

Each temporal logic is based on a particular model of time. The first classification is whether we specify properties which include real-time quantities or simply ordering sequences of events and whether the real-time quantities are integers or real numbers. The second classification is whether we consider time in intervals or simply as points in time. Finally, a third classification is whether we consider time to be linear or branching. Each of these combinations provide different expressive power and therefore different computation complexities for their verification. What follows is a discussion of these classifications with their respective characteristics, advantages and disadvantages.

Dense, Discrete Real-Time and Non-Real-Time Models of Time. Dense real-time means that time can be specified in real-numbers and hence we can refer to any particular moment in time. For example timed automata are based on this dense-time model which is arguably the model closest to the physical world which operated in continuous time [AD94]. This provides a powerful model which allows us to specify features such as liveness, fairness, nondeterminism, periodicity, bounded response and timing delays [AD94]. However, such a model poses a number of challenges as regards to the computability and complexity of the properties that we would like to enforce. The good news is that there are ways to bypass this problem. For example, region automata are used in [AD94] to mimic the actual timed automata. Other literature [HMP92, CP03] propose digitization to solve the problem of handling the dense-time model. This would result in the digitization of the specified system and properties and affectively transforming them into a discrete model of time.
The discrete model of time is very commonly used. Metric temporal logic, PSL and QDDC are all examples of logics based on the discrete time model. It is much more convenient to verify real-time properties which are based on natural numbers rather than real numbers. For example in [HLR92], Halbwachs et al. suggest the use of the language LUSTRE to verify real-time systems by issuing an event for every second. Obviously this can only be done for a discrete time model. A similar approach can also be found in [GHR06]. In practice, although the fact that we are using the discrete time model, seems to be limiting our expressivity, many practical problems involving real-time can still be effectively verified [HMP92].
We can further abstract the notion of time and limit ourselves to time-independent trace properties. Such logics (for example linear temporal logic) have been extensively studied and very efficient algorithms have been proposed [HR02, FS04, GPVW95]. For many practical applications this model offers all the necessary expressivity. Hence, whenever we can limit ourselves to this time model, we will have the benefit of more efficient algorithms.

Interval vs. Non-interval Time Models. Properties can be specified either over points in time or over a set of points in time: an interval. Consider the example: “Within an hour there should never be more than three bad logins” is a property specified on a time interval. Clearly, certain temporal properties are much more easily specified in this manner. Specifying the same property without the concept of an interval would be something like: “At no point should the count of bad logins, starting from the point which is one hour before (the one being considered), exceed three”. The idea of intervals has been proposed by various authors [ZCA91, MM84]. The interesting thing is that introducing very few operators over and above the usual temporal operators, the interval temporal logic becomes much more expressive. A case in point is duration calculus with only two extra basic operators.
However, the notion of intervals in system properties, introduces new complexities for verification. This is because when we specify a property on an interval, the number of sub-intervals in that interval is equal to the number of all possible subsets of time points in that interval. Therefore, if we take a naïve approach in verification, most properties specified on intervals in dense time (such as Interval Duration Logic) are undecidable [Pan02]. However, this problem can be overcome by using particular subsets [Pan02] or some kind of conversion such as digitization [CP03].

Linear vs. Branching Time. Practically every program can have many different traces, where a trace can branch into many different traces. However, it sometimes suffices to consider only one of these branches, i.e. without considering all the other possible branches. Therefore a classification of temporal logics has emerged. This is the distinction between linear and branching temporal logics [Lam80]. In the linear model, a trace of the program is considered as a linear sequence of states where each state has one possible subsequent state. In contrast, in the branching model, at each point in time, all the possible execution paths are considered and thus a computational tree can be generated (where each node may have various possible successors). It has been argued in [Lam80] that linear time is more suited for concurrent programs while branching time is more appropriate for nondeterministic programs. This distinction is however more important in static analysis rather than in dynamic analysis since we can only consider one execution trace (i.e. the one running at the time of verification). Therefore, our focus will be on linear time logic.

2.4.2 Classifications of Logics

Infinite and Finite Trace Logics. An important issue that arises with dynamic analysis is that the available trace which needs to be verified is not a complete one. In other words it is still being created during the execution. Therefore, certain algorithms which work on infinite traces cannot be used without special adjustments. For example when using Büchi automata (whose acceptance condition relies on infinite repetition of a set of states which include a final state), one such possible adjustment is given in [GH01]. However, we won't go into detail about Büchi automata because timed automata have been deemed more appropriate for our work. (A lengthier discussion on automata will be presented in Section 2.4.3.)

A number of temporal logics define their properties on infinite paths. A problem arises with liveness properties. A liveness property is a property which state that *eventually* something should happen. Therefore, it is difficult to verify such a property on traces which are being verified while they are generated. Put differently, while verifying we only have available a finite prefix of a possibly infinite trace. Therefore if an eventuality has not occurred yet, we cannot say that the property does not hold on the actual trace (since we only know its prefix).
To tackle this issue various alternatives have been proposed. Some logics and verification systems introduce new concepts to represent the "maybe" (or "not yet known") state at which we cannot yet decide whether a property holds on a given (incomplete) path [BLS07, EFH$^+$03, Dru00]. For example, in [EFH$^+$03, EF06], the authors provide extra operators over and above the standard temporal logic to include a strong and a weak version for each operator. In the weak version, a liveness property holds if the eventuality has not yet occurred (in a finite trace) while in the strong version it does not. Similarly, in [BLS07] a four-valued semantics is defined which rather than simply true or false, can also return *possibly true* or *possibly false*. Furthermore, Finkbeiner and Sipma in [FS04] proposed the use of statistics to be able to give the actual number of times that eventualities were fulfilled.

Meta-languages. Some of the logics which have been proposed are actually meta-languages which allow the user to specify other logics. For example in the case of EAGLE [BGHS04] and Maude [CDE$^+$99], the proposed architecture provides the basic temporal logic constructs and then allows the user to create his own domain-specific logic.
We investigated two meta-languages: Maude and Eagle. Maude [CELM96, CDE$^+$98] is a metalanguage interpreter which supports equational and rewriting logic computation based on the principle of reflection. Thus, during execution, the specified logic of a system is concurrently rewritten until, possibly, a fully evaluated value is reached. The rules used for rewriting can be considered as transition rules from a computational point of view, or as inference rules from a logical point of view. The rewriting process is in fact continually modifying the system of properties along the execution path; hence the term reflection. This has successfully been used in the runtime verification tool Java PathExplorer [HR01a].

Eagle [BGHS04, GH05] is a runtime verification tool comprising a rule-based language and an interpreter for it. In contrast to Maude, this has been specifically designed to support future and past time logics, interval logics, extended regular expressions, state machines, real-time and data constraints and statistics. Using basic logical operators and two primitive temporal logic operators *next* ($\bigcirc$) and *previous* ($\odot$). Other operators can then be expressed using these primitive operators. It is implemented as a Java library and also allow parametrisation of rules. This means that if for example we would like an acknowledgement to be associated with a particular sent message, we can use the message id to distinguish the corresponding acknowledgement. This provides the logic with expressive power which propositional temporal logic does not have. Another advantage of Eagle is that a lot of its properties can be expressed in state machines which users tend to find more intuitive. Each transition has a condition and an action on the variables of the state. The condition is not only based on the input of the state machine but also on the variables which constitute the state of the machine. Furthermore, the rules expressed in Eagle, can be either maximal or minimal fix-point semantics. This allows more flexibility in expressing weak and strong versions of the same operators. To make this clearer, we will illustrate it using an example. We will define a weak and a strong version of the *until* operator ($Until(F1, F2)$ means that $F1$ has to hold until $F2$ holds). The following is the strong version (which is not satisfied unless F2 becomes true):

$\underline{min}\ Until(FormF1, FormF2) = F2 \vee (F1 \wedge \bigcirc Until(F1, F2))$

The weak version is the same but with $\underline{max}$ instead of $\underline{min}$. All these features make Eagle very versatile for runtime verification and also very interesting for our study which aim to ease the use of runtime verification.

Tool-Specific Logics and Specification Languages. Some systems also propose their own specific language. These have been investigated so that we get an idea of what other designers have found useful to include in their languages. The following is the list which we have explored:

- A simple language is designed in [FH06] which is then easily transformed into timed automata. This is very appealing for our study since we are specifically interested in real-time properties. Apart from this it also uses aspect-oriented programming to weave the verification with the actual system being monitored. This is so applicable to our work that we will give an example of how the architecture works.
  Consider the following two lines:
  $a1 = M.get \rhd start(t); a2$
  $a2 = M.get \rhd \{wait(t, 20); start(t)\}; a2$
  $M.get$ simply means that if the method M.get is called, then we enter the first state which is $a1$. Subsequently, we *start* a timer t and move on to state $a2$. This time, if M.get is called, we have to *wait* for 20 seconds, and then the timer t is re-*start*ed and the state machine goes once more to $a2$. The architecture also supports other timer functions such as *reset* and *cancel*. Another interesting feature is that from one state, we can have a possibility of going into more than one state by adding other entries starting with a specific method call and followed by a clock operation. However, it is important to note that it is assumed that the possibilities are exclusive so that the system remains deterministic. Another suggested way of keeping the system deterministic (but not yet implemented) is by

giving priority according to the order in which the transitions are listed.

- The Policy Specification Language (PSLang) [Erl04] is used in the Policy Enforcement Toolkit (PoET). In PSLang, the security state is stored in named and typed variables. This makes the system more transparent to the user because there is no hidden variables he is not aware of. Furthermore, it is able to use low-level actions to synthesize higher-level security events according to the specified policies. Consequently, these events can trigger the required enforcement activities. This hierarchy of actions and events makes it possible to make policies reusable and more clear without unnecessary details. PSLang has been specifically designed to be easily used and thus the syntax and semantics are based on Java and JVML. In fact as regards the syntax we found it to be very similar to aspect code. However, it has the extra overhead of declaring a lock and longer syntax.

- ConSpec is inspired by PSLang, but is however more restricted [AN07]. It is particularly intended for securing mobile devices with limited resources. A contract is defined for each application and upon installation on a device, the contract is checked against the user's (of the device) policies. If the application's contract does not comply with the user's policies, the application cannot be installed on the device.

- Polymer was developed to help users enforce security policies on untrusted Java applications [BLW05]. A Polymer policy is implemented by extending the *Policy* object which contains decisions (queries) and actions, security state of the running application and methods to update the policy's security state. Furthermore, policies can be used to compose higher-order security policies which in turn will combine the sub-policies in a semantically meaningful way.

- Tempura [Mos86] has been proposed specifically to describe interval temporal logic. An example of a Tempura formula is as follows: $(M = 4) \wedge (N = 1) \wedge halt(M = 0) \wedge (M gets M - 1) \wedge (N gets 2N)$. This means that in the current state, M should be equal to 4 and N should be equal to 1. Then for the interval to satisfy the formula, in the next state, M should be equal to 3 (because $M gets M - 1$) and similarly N should become 2. Subsequently, in the next state of the next state the new values of M and N should hold until the *halt* condition is met. If all the states within the interval satisfied the resulting values of M and N, then the interval satisfies the formula.
  Another such logic is the Allen temporal logic [All84]. It is interesting to note that it is possible to transform Allen temporal logic into linear temporal logic in linear time [RB06]. The advantage of using such an interval logic is that it is very appropriate to represent constraints in fields such as project planning where a number of events occur, each occupying a time interval [RB06].

- PROMELA (Process Meta Language) is the verification language used in SPIN [Hol97] to specify the design of a system (to be verified) without implementation details. Furthermore, the correctness requirements of the system are specified in linear temporal logic. Subsequently, SPIN performs model checking to ensure that the design specifications of the system are consistent with the correctness properties.

- The Primitive Event Definition Language (PEDL) and Meta Event Definition Language (MEDL) used in the Monitoring and Checking (MaC) architecture [LKK$^+$99] are two complementary languages which allow for a clear separation between the definition of the primitive events of a system (using PEDL) and the system properties (using MEDL) possibly taking into account various primitive events. The motivation for this separation is that it makes the system easier to adapt to other programming languages, leaving the higher level definition language (MEDL) intact. For example using PEDL we may specify a primitive event as follows:
  ```
  Event OpenGate = StartM( GateController.open() );
  ```
  Then, using MEDL we write:
  ```
  Cond GateClosing = [ CloseGate when !Gate\_Down, OpenGate)
   => lastClose + 30 > currentTime;
  ```
  The example is a simplified version of the one in [LKK$^+$99], but it is enough to understand that in MEDL we are using primitive events and condition to check that the time being taken by the gate to close is not longer than 30 time units.

- PSL (Property Specification Language) [EF06] allows the user to specify stand alone properties (i.e. which are not part of the actual system code) which are mathematically rigorous and automatically verifiable. An interesting aspect of PSL is that it provides two versions of the temporal operators: the weak and the strong. The purpose of this is to provide the user with two different ways to handle instances during runtime where the system does not have the whole trace and hence it cannot provide the final conclusion. Therefore, there are two possible approaches to take when we still lack a final conclusion: either consider it as a positive because the negative has not happened (weak version), or consider is as a negative because the positive has not happened (strong version). For example the rule $eventually(p)$ in a trace being executed where p has not yet occurred, can be considered to be true in the weak version or false in the strong version. Furthermore, PSL also includes other features such as Sequential Extended Regular Expressions and clocks.

Other Logics. The literature is full of proposed formal notations for specifying system properties. Different authors have suggested different specification notations according to the particular domain, trying to improve one aspect or another of the existing notations. An important tradeoff that should be highlighted is that the greater the expressivity of the formal notation used, the more complex is the algorithm for its verification.

- We will first start by the basic linear temporal logic proposed by Pnueli (as reported in [Lam80]). In linear temporal logic we define properties which hold on a sequence of states without branching (unlike computational tree logic (see Section 2.4.1)). This is ideal when we consider a single execution path as in dynamic analysis. An example of a property which can be specified in linear temporal logic is $G(p)$ or $\Box(p)$ which states that p should hold for all the states in the sequence. We can restrict linear temporal logic to consider only past states in an execution trace. The resulting logic is known as past time linear temporal logic. In this case rather than considering the all the states in a trace, we consider only the past states. Similarly, we can consider only the future states using future time linear temporal logic. In various cases it is much more convenient to express certain temporal properties using past time linear temporal logic rather than future time linear temporal logic even though they have the same expressive power [HR04]. Another variation of linear temporal logic is Finite Trace Linear Temporal Logic [HR01c]. This is an adaptation of linear temporal logic which rather than considering an infinite execution trace, is applied on a trace with a finite number of states. This is especially useful for dynamic analysis (as opposed to static analysis (see Section 2.3)), since when verifying at runtime we only have a part of the execution trace. Linear temporal logic syntax is quite easy to understand as a logic, but as formulas become complicated, it may not be so straight forward to understand the meaning. This is especially true for developer who are not familiar with logic. Furthermore, it is not expressive enough to represent real-time properties.

- The metric temporal logic was introduced [Koy90, CMP94] to add the necessary expressive power on linear temporal logic to represent real-time properties. Metric temporal logic has been successfully used in Temporal Rover with some extensions [Dru00]. Basically, it uses the same notation as linear temporal logic and adds real-time quantities as additional constraints. Furthermore, metric temporal logic has been studied as regards to its expressiveness and complexity [AH93]. Interestingly, metric temporal logic with the past temporal operators is expressively complete and yet elementary decidable. Since metric temporal logic is based on linear temporal logic, the same problem with the readability/understandability of non-trivial formulas arises.

- Regular expressions have been suggested as an extension to the MEDL language [SS03] in the MaC architecture. The motivation of adopting regular expressions is that they are more convenient to express certain complex orderings of events. In fact a considerable number of temporal logics and architectures have been specifically designed or extended to include regular expressions' expressiveness. Examples include: ForSpec Temporal

Logic (FTL) [AFF$^+$02], the logic supported in monitoring-oriented programming [CR03], PSL [EF06] and other logics such as QDDC [Pan01] have been shown to be able to encode regular expressions. A very positive aspect of regular expressions is that they are used for other applications such as string matching and therefore developers are already familiar with its meaning. For example imagine we want to denote a simple rule: any number of occurrences of event a can occur before event b. Using regular expressions this can be written simply as: $(a^*)b$. Using another logic such as linear temporal logic it can be written as:
$AsThenB(\varphi_n) \overset{\text{def}}{=} b \vee (a \wedge Next(AsThenB(\varphi_{n-1})))$
This can be much less easy to come up with, especially for someone who is not accustomed to temporal logics. However, regular expressions also have their drawbacks and may not be intuitive even for someone who is accustomed to use regular expressions in other domains. For example, upon seeing a regular expression made up of system events, the user may get confused whether the represented sequence is an accepted or a non-accepted sequence of events.

- Duration calculus, introduced in [ZCA91], is an interval logic which we found to be very elegant and succinct. It has only two extra basic operators: "chop" ($\frown$) (which catenates two intervals) and the integration symbol ($\int$) (returning the size of the interval which satisfies a certain property). Using these two simple operators we can define very useful properties such as: $\Box(badLoginCount > 3 \Rightarrow length > 60 mins)$. This simply means that for every sub-interval (hence the *Box*), if the variable badLoginCount exceeds three, the length of the interval should be longer than an hour.
  However, the interval notion of time introduces new complexities for verification. This is because when we specify a property on an interval, the number of sub-intervals in that interval is equal to the number of all possible subsets of time points in that interval. Therefore, if we take a naïve approach in verification, most properties specified on intervals in dense time (such as Interval Duration Logic) are undecidable [Pan02]. However, this problem can be overcome by using particular subsets [Pan02] or some kind of conversion such as digitization [CP03].

2.4.3 Automata in Verification

Automata have been extensively used in verification especially for efficiently deciding whether a property is violated at a particular state. A very attractive advantage of using automata is that they are a pictorial representation. Since we intend to make our architecture "easily" usable for developers this representation may be more intuitive than other textual representations of the security properties.

- In the case of regular expressions as used with MEDL [SS03], they are first converted into a finite state automaton. This is then used for the actual verification algorithm. Finite state automata are proposed in [RB06] to efficiently monitor temporal properties written in Allen Temporal Logic (ATL). Automata are also proposed to check finite traces in [GH01]. A lot of literature provides conversions for particular notations to automata and therefore, automata may be the ideal candidate if we need to integrate different notations together.

- Büchi Automata has been suggested to check finite traces by generating the corresponding labelled generalised Büchi automaton (LGBA) from linear temporal logic [DFRR04, GPVW95]. A similar approach is used in [Bod05] and [CVWY92] where the Büchi automata are used for efficient verification of temporal properties. Also, the properties specified in the model checker SPIN [Hol97] are converted to Büchi automata. Subsequently, the whole system together with the specified properties becomes the asynchronous interleaving product of automata. However, a number of problems have been identified in [RH01] which arise from the use of Büchi automata. Most notorious are the problem of converting linear temporal logic formulas into Büchi automata and secondly, the problem of checking finite traces when Büchi automata are thought to handle infinite strings. This explains why a number of other verification systems [RH01, LKK$^+$99, Dru00] do not use Büchi automata.

- Alternating Finite Automata (AFAs) have also been very commonly suggested for the verification of temporal properties [Dru06, SB06, FS04]. The advantage of using AFAs is that it has *and*-states and *or*-states which can be exploited to represent the recursive definitions of linear temporal logic which also involves *and*s and *or*s. Another advantage mentioned in [Dru06] is the visual appeal of AFAs and more importantly, that AFAs are linear in size to the corresponding linear temporal logic formula [FS04]. Once we have the AFA equivalent of the linear temporal logic formula, the AFA is reconfigured according to each online input, minimising the AFA where appropriate.

- Timed Automata [AD90, AD94] have been repeatedly proposed for verifying real-time temporal properties. On the transitions, apart from an event, we can define clock operations. Figure 1 shows a simple practical example of a timed automaton.

  It represents the logic required to ensure that a gate closes 1000 time units. So as soon as a *Close_gate* event is received, a timer is reset to zero. If 1000 time units elapse and the gate is still closing, then we proceed to an alarm state. While, if the gate is closed within the given time, then we proceed to an accepting state.
  The literature contains various applications where timed automata where used. In [FH06], the architecture involves the translation of both the system and its properties into separate timed automata. Subsequently, these are both weaved together to produce a single timed automaton. This is then optimised and used to verify whether the specified properties

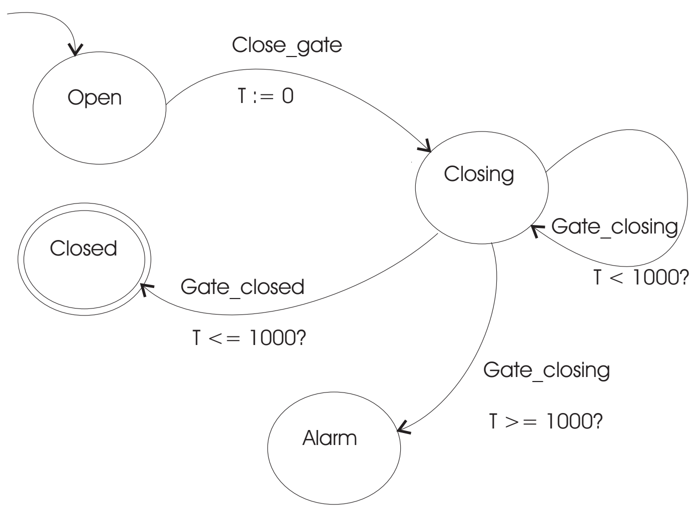

Figure 1: An example of a timed automaton which handles the closing system of a gate.

hold. The advantage of this approach is that the user can visually see the resultant weaved system and can also control it. Other instances of using timed automata are [STY03, Bou06]. Interestingly, Bouyer in [Bou06] adds weights as an extension of timed automata with costs which is especially useful for simulating resource consumption in timed systems.

- Mode automata [MR03] have the purpose of providing a two-level automaton (multi-level if a composition is used). The motivation behind this structure is the need to separate logic which runs at a particular instant (say only during take-off) from other logic that runs at another instant (say only during landing). The need of separation occurs frequently in many applications for example mode automata may prove to be very useful in separating the verification of states in a transaction from the internal logic necessary for each state. Hence each state represents a possible mode in which we can be.

2.5 Instrumentation Approaches

The monitoring and verifying code require that somehow, they are instrumented with code of the system being verified. There are various ways of achieving this. In Temporal Rover the code is inserted in the actual system's code as if the logic is part of the system [Dru00]. This can be considered as if the instrumentation is being done manually. In other cases such as in MAC [LKK+99] and PathExplorer [RH01] the instrumentation has a higher level of automation since the specification is separated from the actual system's code. We do not want to leave the instrumentation to be done manually, because this is error-prone and we do no want our error checking mechanism to be error-prone! Therefore, it is desirable to find a way to automate the instrumentation of the monitoring code [CM05]. Aspect-oriented programming seems to be a very good option.

Aspect-Oriented Programming used for Runtime Verification. The concept of using aspect-oriented programming in runtime verification as a means of instrumentation is far from new [OGRG07, dH05, SB06, Bod05, JM07, GSSP02, CF00, FH06]. However, there are various levels and ways in which aspect-oriented programming can be used. For example using AspectJ [KHH$^+$01], the user can write the monitoring code in a separate aspect, leaving the code of the actual system clean from monitoring code and at the same time, the code regarding the monitoring is all concentrated in one place. For example, imagine we want to stop (or warn) users of a code library in the case of wrong usage. The rule which should be verified is that the library should be initialised before any other method is used. The code which blocks any method call before initialisation is shown in Listing 1.

Listing 1: Library checking advice

```
Object around():(execution (* Library.*(..)) && !execution(*
Library.initialization(..))) {
    if (initialized)
        return proceed();
    else
        return "LIBRARY: Library must be initialized.";
    }
```

Using the “*” wildcard, we have managed to check for initialisation before the execution of any possible method apart from the one whose name is *initialization*. One should note the efficiency of implementing such logic in a few line of code rather than inserting a condition at the start of all the methods in the library. Furthermore, adding further methods to the library does not necessitate any modifications to the code handling the property.
Having automatic code injection and all monitoring code in one aspect is much less error-prone, but sill requires a lot from the developer and therefore there is still a lot of possibility for error. Hence, it has be suggested [dH05, Bod05, SB06, OGRG07] to create an automatic way to generate AspectJ directly from the system's specification.
A slightly different approach is proposed in [CF00] where the specifications are directly weaved into the graph representing the program without going to the intermediate stage of creating aspects. Similarly, in [FH06] is to specify the properties in a special language semantically equivalent to timed automata and the program to be verified is also abstracted into timed automata. Subsequently, the weaving is performed on timed automata. The advantage of this approach is that the weaving is visible to the programmer. Furthermore, in specifying the properties the programmer can focus on the semantic meaning of the properties rather than the syntax as in the case with using a specific aspect-oriented programming language (such as AspectJ).

2.6 Algorithms for Verification

To start this section we must first make a clear distinction that some algorithms are aimed to verify all the possible execution traces while others need only verify one execution trace. It is clear that it is much more challenging to verify all the execution traces rather than a single execution trace. Further to this, we must also distinguish between verifying infinite traces as opposed to finite traces which we obtain during runtime. In this section we will give an overview of the various approaches taken to solve these variations of the verification problem. Another question which would like to ask about our system may be whether a property is decidable or not. In other words, the decidability question asks: "can we decide wether this property always (for all execution paths) holds or not?" In fact this is the equivalent of the previous satisfiability question. Satisfiable means that there is at least one execution path which satisfies that property. This is because if the negation of a property is satisfiable, then it surely does not hold for all the paths. The decidability issue for temporal logic was considered in [BG85] and it is shown that temporal logic of real order is decidable.

Automata-Based Algorithms. Trace checking is the process of verifying that a property holds throughout a particular trace. A possible approach to trace checking is to translate the formulae into automata and then run the automaton in parallel to the system until a satisfying state is reached [Hol97]. A number of different automata used for this purpose have already been outlined in Section 2.4.3. To do this efficiently a tableau-based algorithm is usually used [GPVW95, DFRR04, KMMP93, GD00, Gei03] in which not all the nodes are constructed from the beginning, but instead they are generated incrementally as required. This approach has been used extensively to check properties on all the possible execution traces. However, adding certain extensions to temporal logic may in fact render it undecidable (in a tractable amount of time) [AH93, SC82]. Vardi and Wopler in [VW86] propose another algorithm which they claim to be cleaner than the tableau-based algorithm. The argument is that they separate the construction of the automata (which correspond to the formulae) from the actual verification of the program (by relating the automaton to the program).

On-the-fly Algorithms. Another subset of algorithms are those which divide the predicates to be verified into two parts. The basic idea is that since we do not have the whole trace available during runtime verification we must satisfy the formulas as we progress through the execution. Hence, we split the formula into two parts: the part which we should check "now" and the part which we will check later; hence the name "on-the-fly". For example verifying a formula having the $\Box$ (*always* operator) will entail the verification that the formula holds at the current state and also that it will hold in the next state. Similarly, a formula having the $\diamond$ (*eventually* operator) will require a test of whether or not the formula holds at the current state and if it does not, it has to be checked again in the next state. Such algorithms exploit the recursive nature of the temporal logic and use the results from the consecutive states to

reach the result of the current state. Using automata to represent formulas, we will in fact be generating the verifying automaton on-the-fly according to the previous state. It is sometimes necessary for automata to be generated on-the-fly, either because the automaton is infinite, or because it is too large to fit in memory. This is the case with symbolic automata; they are infinite. The reason is that symbolic automata allow the use of variables, which can take an infinite number of values (in theory). Therefore, it is imperative that whenever we use such automata for verification, we use an on-the-fly verification algorithm.
The idea behind dynamic programming is the very same idea of reaching a result based on the previous results. In fact, dynamic programming has been suggested as a very efficient way of checking whether a formula hold or not at a certain state of the execution [RH01]. This algorithm exploits the fact that the properties can be recursively defined and hence there is no need to test the whole trace at each point in time; it is sufficient to consider the current state while having the computed result of the rest of the trace. Thus this give a time complexity of $\mathcal{O}(n)$ The main drawback is that the trace has to traversed backwards and hence cannot be used to monitor a trace during its execution. However, later on the algorithm was implemented to perform verification at runtime [HR02] and thus the drawback was overcome.
Another suggested similar approach is rewriting [CELM96, HR01c, HR01b, HR04]. In this case, a formula is transformed (reduced) after each event (hence the word rewriting) until it is satisfied or violated at a later state in the trace. Very similar work has also been done in [BGHS04] where the "temporal formula can be separated into a boolean combination of pure past, present and pure future".

2.7 Overheads in Using Dynamic Analysis

Since dynamic analysis involves some kind of monitoring and extra verification, then it obviously introduces an overhead to the system being monitored. Such an overhead can many negative consequences on the system which must react under real-time constraints. The approach to tackle such a problem is two-fold. On the one hand we would like to minimise the overhead and therefore try to optimise the verification process, while on the other hand, we can try to give guarantees as to the amount of processing and memory resources which the verification process would require.
The kind of optimisations which are possible are very specific to the architecture used and at which phase of the verification process this is carried out. For example when using automata, there is the concept of pruning [FS04] or collapse [Dru06].

Other proposed optimisations try to avoid unnecessary checking during runtime [FH06, UES00]. In [CF00], Colcombet and Fradet propose a mixture of static and dynamic analysis so that properties which can be verified upon inspecting the source code are not unnecessarily tested for during runtime. Thane in [Tha00] classifies the various types of monitors. Furthermore, for each type Thane explains how this can or cannot be removed from the target system.

Havelund and Roşu in [HR04] suggest to optimise boolean functions and to evaluate the predicates which probabilistically add minimum runtime cost.

Drusinsky in [Dru06] analyses the growth of p-trees under different restrictions. In this way guarantees can be given as to the upper-bound of the size of the tree (and hence memory usage). To save memory usage, [CVWY92] proposes the use of hashing without collision detection to store program state with the disadvantage of possibly missing some states.

In case studies carried out in [BGK$^+$02], the overhead of runtime verification is quite large. Hence, two kinds of optimisations (abstractions) were proposed in the scenario of a packet routing algorithm. One is that the verification was limited to a certain number of nodes rather than all the nodes (population abstraction) while the other limited the verification to a particular type of packets (packet-type abstraction). However, these optimisations are reasonable if the aim of the verification is that of finding errors before the actual deployment rather than that of ongoing monitoring after the target system has been installed.

3 Proposal

3.1 Synchronous Programming, Reactive Systems and Runtime Verification

When we take the hassle to employ runtime verification we do so for systems where errors are highly undesirable. The simple applications which we daily use are not usually verified but software controlling railways or other expensive machinery is. Most systems which we would like to verify fall under the class of reactive systems. Such systems react to inputs from their environment by changing some outputs at real-time. A class of languages which have been specifically designed for programming reactive systems are known as synchronous languages. The synchronous nature of these languages lies in the fact that the reaction time of the system is considered to be negligible i.e. the system reacts instantaneously to the inputs. One such language is LUSTRE [HCRP91]. A very important advantage of LUSTRE is that the memory required for the monitoring code is measurable at runtime. This is very important in security-critical systems since it enables us to give guarantees on the upper bound of memory required by the monitoring overhead. A small example of LUSTRE code is shown in Listing 2.

Listing 2: Access checking node

```
node BadAccess(w,r,i,o:bool)returns(bw,br:bool);
var l:bool;
let
  l =    if (o) then false
     else if (i) then true
     else false->pre(l);
  bw = w and not(l);
  br = r and not(l);
tel
```

What this piece of code does is that it monitors four events: write (w), read (r), login (i) and logout (o). It keeps track of whether a user is logged in or logged out in the local variable l. If a read or write event occurs while login (l) is false, br (or bw respectively) is set to true.
Another interesting point of view is that LUSTRE can be considered as an executable temporal logic [HLR92]. This has motivated the specification of both the system and its properties to be in LUSTRE in [HLR92]. A similar concept introduced in [HLR93] is the concept of *observers*. An observer is a program which checks that the main program keeps to its specification. Hence, the verification problem then simply involves the composition of the two programs and ensuring that the observer never reaches an undesired state. This concept has been extended some years later in [Ray96] adding the expressive power of regular expressions. More specifically, properties including the regular constructs: sequence and iteration are translated into an equivalent Boolean dataflow network in the language LUSTRE. Another recent development was the synthesizing of non-deterministic LUSTRE observers from QDDC [GHR06]. Recall that QDDC is a very expressive interval logic based on the discrete model of time. Furthermore, a deterministic subset of QDDC was used to show how this can be verified using deterministic observers. Imagine we want to implement the $\sum Q$ QDDC operator which counts the number of time Q was true during a particular interval. Using the simple LUSTRE code in Listing 3, we can count the number of times q was true since the start of a particular period (indicated by p being true).

Listing 3: Access checking node

```
after_p = p or (false -> pre(after_p));

nb_q_since_p = if p then (if q then 1 else 0)
                  else if after_p then
     (pre(nb_q_since_p))+(if q then 1 else 0)
                  else 0
```

3.2 Discussion

After considering a lot of related work, we now move on to discuss possible ways of tackling the proposed project.

3.2.1 Dynamic Analysis Flavour

First we have to choose the flavour of dynamic analysis which we will use. The assertions in design by contract are made manually in various locations in the system code. This immediately introduces the possibility of errors since humans can easily omit or misplace some assertions. Ideally, the assertions are not manually inserted into the system, but rather automatically weaved. Moreover,

we would like the security properties to be centralised rather than scattered throughout the code. Changing a security property from a central location is much easier and less error-prone. Considering these issues in design by contract, we will not adopt this approach in our research.
Runtime verification is much more appropriate for our research because it uses formal notation for specifying security properties which is more succinct and abstract than the actual implementation. Another attractive aspect of runtime verification is that it allows the user to specify extra code so that the system which finds itself in a bad state, can be reverted back to a valid state. This is very desirable because it eliminates the need of human intervention upon a security violation. This will be further investigated in our research, because developers may not like a verification system to be intrusive (This will be discussed in Section 3.3). We also intend to explore the possibility of off-line verification if we find the verification overhead to be too large.
Exploring the possibility of having the violation mechanism as part of the system design, as designated by monitoring-oriented programming, is a very interesting area. However, in our research we plan to limit ourselves to treat the verification as a "double-check" rather than the actual check. This decision is mainly due to the fact that the systems under our consideration have already been designed and implemented. Furthermore, there is the issue of synchronous and asynchronous verification which has already been mentioned as online versus off-line verification. However, we can also explore the possibility of finding a compromise between total synchronicity and completely off-line verification. This can be achieved by allowing a possibility of a delay between the verification system (which can be run on a separate machine) and the target system. Furthermore, the delay allowed may be dependant on the criticality of the part of the system in which we are executing. Therefore, in a critical part we may wait for the verifier to synchronise, while in a non-critical part we can afford to allow the verifier to run asynchronously.
The idea of having an explanation of how a bad state was reached, as suggested in runtime reflection, is very desirable. This is more especially so when dynamic analysis is used during the testing phase to identify errors. In our research we have to provide a means for the user to understand what went wrong in the system. This is further elaborated in Section 3.

3.2.2 Static Analysis vs Dynamic Analysis

It would have been very interesting to integrate static analysis and dynamic analysis in one system. For example static analysis could have been very useful to intelligently generate test cases for a dynamic analysis tool to find errors during testing [ABG$^+$05]. However, we have to focus our research because of many limitations and hence we have to keep to dynamic analysis.

3.2.3 Logics

In our research we will first start to consider sequences of events without real-time constraints. This will make it easier to implement verifying algorithms. However, later on we intend to provide more expressivity as the need arises so that we will be able to handle properties which include real-time. Having said this, we intend to be very careful not to provide unnecessary expressivity which will only result in more complex verification algorithms without adding any benefit. To this end, we will explore different formal notations and if necessary even create our own. This will be further discussed in the proposal in Section 3.

Choosing the logic which is most appropriate for our project is not trivial. Linear temporal logic is arguably very easy to understand as a logic but it is not expressive enough for our intents and purposes, since we have the target of verifying real-time properties. Hence it was not further explored. Furthermore, to express certain complex rules in linear temporal logic is not trivial. On this ground, regular expressions may be much more desirable because they are so widely used in other applications and therefore users may already be familiar with them. We have also investigated the possibility of introducing a real-time extension over regular expressions and found out that this has already been done in [ACM02] with timed regular expressions. Apart from regular expressions, we also intend to explore the possibility of using duration calculus because of its elegance and expressiveness. However, we will most probably stick to a fragment of it so that it remains relatively simple to verify algorithmically. Since we intend to provide a whole framework of notation, to allow flexibility of choice to the user, we need some way of integrating the notations together. regular expressions are known to be very easily translatable into finite state automata. Furthermore, we are interested in timed automata since these provide us with the necessary expressive power for representing our security properties with real-time constraints. However, we still need to explore the relationship of timed automata to the other notations which we intend to use, including timed regular expressions and duration calculus. Providing the user with the choice of using any of these notations and being able to seamlessly integrate the various properties would be very desirable.

3.3 Proposed Project Directions

The proposed direction of the project is two-fold:

- To find a suitable case-study and create a user-friendly system which allows a user to declare a number of relevant events which can be extracted from the system under consideration and a number of properties which the system should adhere to. Then our artefact should generate the necessary monitoring and verifying code so that this will both extract the relevant events and also ensure that these events are not violated at runtime.
  A very important idea which complements the idea presented by Leucker

and Schallhart in [LS07], is that our architecture should be able to distinguish among the various violations that are possible. Furthermore, this allows the user to specify actions which the (generated) monitoring system will perform on his behave if a particular violation is encountered. This issue is quite delicate since this may interfere with the target system being monitored. There is a spectrum of ways in which the verification system can help the user if a violation is detected. At the lower end there is a simple logging of the problem, while at the higher end the system can intervene and stop the running system altogether for safety. Therefore the level up to which the monitoring system will intrude with the running of the system should be left totally in the user's jurisdiction.

- The second is to explore different logics and notations to find the most appropriate one or ones. The important aspects of a specification logic are: expressivity, ease of use from the user's perspective and easy to verify in terms of algorithmic complexity. Balancing these aspects is of utmost importance. Hence, we will refrain from introducing any unnecessary expressive power so as to keep the notation as clean as possible and at the same time minimally complex to verify. We will also consider the possibility of allowing the user to use a variety of different notations which can then be amalgamated together in some framework.

3.4 The Implemented Prototype

A simple prototype has already been implemented with the aim of demonstrating the application of our ideas. A simple target system implemented in java has allows the user to perform four different activities: login, logout, read and write to a database. The simple property which we wanted to verify was that no read or write should occur unless the user is logged in. A text file was used to enter the necessary information required for the verification. The text file contained three sections. The first part contained a list of events defined over a number of method calls. Then, the second part contained a LUSTRE node which took the predefined events as inputs. The outputs of the node represent possible violations which can be detected. In this way we can distinguish among the violations which are possible. The final section included a reaction and a message to be logged for each case possible property violation. Our prototype tool took the text file as an input and generated two java classes and two java aspects. The first java class is the Verifier which runs on a separate thread (possibly on a separate computer system with some modifications of the communication channel) which receives a stream of events and checks whether any of the properties (written as LUSTRE in the text file) are violated. The second java class is a Listener which handles the communication between the system being verified and the verifier. Furthermore, the two java aspects have the purpose of injecting code in the target system: one injects code for monitoring and eliciting events and the other to inject reaction code (to be called if a property violation occurs).

3.5 A Case-Study

Another very interesting development in the project is a real-life case-study which is being carried out in relation to a local company which depends on highly secure software. This experience is proving to be very enriching both to the industry and also from an academic perspective. Throughout the case-study we will try to extract the security properties from the specification of the system under scrutiny and then use an appropriate formal notation to represent these properties. The choice of the notation will be based on discussions with the developers, testers and security personnel of the company so that the notation is as easy to use as possible but at the same time has the required expressivity.

4 Conclusions

As our society becomes more reliant on computer systems, the need for reliable software is ever increasing. A compromise between testing and static analysis is dynamic analysis. This involves the specification of security properties which are then verified while the system is executing. The advantage of dynamic analysis is that it is scalable (unlike static analysis) and verifies the actual trace being executed (unlike testing). However, instrumenting the system code with monitoring and verifying code is error-prone. Therefore, automating this process is very desirable. There are various design choices in implementing dynamic analysis. Prominent, is the choice between on-line and off-line verification. This really depends on the specific purpose for which the verification is used (for debugging and testing or during actual deployment of the system) and on how much we can interfere with the monitored system.
Another design issue is the choice among various logics and languages which were proposed to describe security properties for dynamic analysis. There are various kinds of logics which can be classified under a number of headings. There are real-time and non-real-time logics, infinite trace logics and finite trace logics, tool-specific and non-tool-specific, etc. Furthermore, there are also various kinds of automata, which have also been used to specify properties to be verified at runtime. Their advantage over textual logics is they they are more pictorial in nature and various algorithms have already been devised to resolve satisfiability and decidability in automata.
Apart from a way of representing the security properties, we need the mechanism to instrument the system code with the monitoring code. A possible means of automatic instrumentation is aspect-oriented programming. This approach provides a modular way of representing a cross-cutting concern such as security. There it is very attractive for our intents and purposes.
Another important consideration in introducing dynamic analysis is the overhead it creates over and above the system. This is inherent in the fact that the system is being verified while it is executing. Various measures are proposed in literature, aimed at mitigating the effect of this overhead. A possible approach is to give an upper-bound guarantee of the amount of memory required for the

verification overhead during runtime. To this end the synchronous language LUSTRE can be useful because we can know the amount of memory required at compile time.
In the light of the literature review, we intend to find a real-life case-study and experiment with the dynamic analysis approach. The main aim is to make the specification of the system security properties as user-friendly as possible. This should make the engineering of correctness and reliability more easily integrated into the commonplace development of systems.
The vast literature in the area of runtime verification is a proof of the growing interest in the subject. Furthermore, the subject is still relatively new and there is a lot of space for research. The ideas which we have proposed so far have already brought up a lot of interest from the local industry and this is a very positive sign. We are very optimistic that the research will evolve with a lot of more innovative ideas and possibly provide practical solutions for current security issues.

References

[ABG$^+$05] Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Mike Lowry, Corina Pasareanu, Grigore Rosu, Koushik Sen, Willem Visser, and Rich Washington. Combining test case generation and runtime verification. *Theor. Comput. Sci.*, 336(2-3):209–234, 2005.

[ACM02] Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *J. ACM*, 49(2):172–206, 2002.

[AD90] Rajeev Alur and D. L. Dill. Automata for modeling real-time systems. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 322–335, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

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

[AFF$^+$02] Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, and Yael Zbar. The forspec temporal logic: A new temporal property-specification language. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 296–211, 2002.

[AH93] Rajeev Alur and Thomas A. Henzinger. Real-time logics: Complexity and expressiveness. *Inf. Comput.*, 104(1):35–77, 1993.

[All84] James F. Allen. Towards a general theory of action and time. *Artif. Intell.*, 23(2):123–154, 1984.

[AN07] I. Aktug and K. Naliuka. Conspec: A formal language for policy specification. In *FLACOS '07*, pages 107–109, Oslo, Norway, October 2007.

[BG85] John P. Burgess and Yuri Gurevich. The decision problem for linear temporal logic. *Notre Dame J. Formal Logic*, 26(2):115–128, 1985.

[BGHS04] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *VMCAI*, pages 44–57, 2004.

[BGK$^+$02] Karthikeyan Bhargavan, Carl A. Gunter, Moonjoo Kim, Insup Lee, Davor Obradovic, Oleg Sokolsky, and Mahesh Viswanathan. Verisim: Formal analysis of network simulations. *IEEE Trans. Software Eng.*, 28(2):129–145, 2002.

[BLS07] Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In O. Sokolsky and S. Tasiran, editors, *Proceedings of the 7th International Workshop on Runtime Verification (RV)*, volume 4839 of *Lecture Notes in Computer Science*, pages 126–138, Berlin, Heidelberg, November 2007. Springer-Verlag.

[BLW05] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with polymer. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 305–314, New York, NY, USA, 2005. ACM.

[Bod05] Eric Bodden. Efficient and expressive runtime verification for java. Grand Finals of the ACM Student Research Competition, 2004/2005.

[Bou06] Patricia Bouyer. Weighted timed automata: Model-checking and games. *Electronic Notes in Theoretical Computer Science*, 158:3–17, 2006.

[CDE$^+$98] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude as a metalanguage. In *In 2nd International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[CDE$^+$99] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. The maude system. In *RTA*, pages 240–243, 1999.

[CDR04] Feng Chen, Marcelo D'Amorim, and Grigore Roşu. Monitoring-oriented programming: A tool-supported methodology for higher quality object-oriented software. Technical Report UIUCDCS-R-2004-2420, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, 2004.

[CELM96] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of maude. *Electr. Notes Theor. Comput. Sci.*, 4, 1996.

[CF00] Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66, 2000.

[CM05] Séverine Colin and Leonardo Mariani. *Model-Based Testing of Reactive Systems*, volume 3472. Springer Berlin / Heidelberg, 2005.

[CMP94] Edward Y. Chang, Zohar Manna, and Amir Pnueli. Compositional verification of real-time systems. In *Logic in Computer Science*, pages 458–465, 1994.

[CP03] Gaurav Chakravorty and P.K. Pandya. Digitizing interval duration logic. In *Proc. CAV 2003*, Colorado, Boulder, July 2003. (Technical Report, TCS-02-PKP-1, Tata Institute of Fundamental Research, 2002).

[CR03] Feng Chen and Grigore Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *RV'03*, volume 89(2) of *ENTCS*, pages 108 – 127, 2003.

[CVWY92] Constantin Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[DFRR04] Rocco Deutschmann, Matthias Fruth, Horst Reichel, and Hans-Christian Reuss. Trace checking with real-time specifications. In *Proceedings of the 5th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/-FORMAT 2004)*, Braunschweig, Germany, December 2004.

[dH05] Marcelo d'Amorim and Klaus Havelund. Event-based runtime verification of java programs. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

[Dru00] Doron Drusinsky. The temporal rover and the ATG rover. In *SPIN*, pages 323–330, 2000.

[Dru06] D. Drusinsky. On-line monitoring of metric temporal logic with time-series constraints using alternating finite automata. *Journal of Universal Computer Science*, 12(5):482–498, 2006.

[EF06] Cindy Eisner and Dana Fisman. *A Practical Introduction to PSL (Series on Integrated Circuits and Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[EFH$^+$03] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In *cav03*, volume 2725 of *lncs*, pages 27–39, Boulder, CO, USA, July 2003. springer.

[Erl04] Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement.* PhD thesis, Cornell University, 2004.

[Ern03] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, OR, May 9 2003.

[FH06] P. Fradet and S. Hong Tuan Ha. Systèmes de gestion de ressource et aspects de disponibilité. *Revue francophone L'Objet*, 12(2-3), 2006.

[FS04] Bernd Finkbeiner and Henny Sipma. Checking finite traces using alternating automata. *Form. Methods Syst. Des.*, 24(2):101–127, 2004.

[GD00] Marc Geilen and Dennis Dams. An on-the-fly tableau construction for a real-time temporal logic. In *FTRTFT '00: Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 276–290, London, UK, 2000. Springer-Verlag.

[Gei03] Marc Geilen. An improved on-the-fly tableau construction for a real-time temporal logic. In *CAV*, pages 394–406, 2003.

[GH01] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *Automated Software Engineering*, 2001.

[GH05] Allen Goldberg and Klaus Havelund. Automated runtime verification with eagle. In *MSVVEIS*, 2005.

[GHR06] Laure Gonnord, Nicolas Halbwachs, and Pascal Raymond. From discrete duration calculus to symbolic automata. *Electr. Notes Theor. Comput. Sci.*, 153(4):3–18, 2006.

[GPVW95] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[GSSP02] Andreas Gal, Olaf Spinczyk, and Wolfgang Schrder-Preikschat. On aspect-orientation in distributed real-time dependable systems. In *WORDS*, pages 261–270, 2002.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[HLR92] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. *IEEE Trans. Softw. Eng.*, 18(9):785–793, 1992.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[HMP92] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. What good are digital clocks? In *ICALP '92: Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, pages 545–558, London, UK, 1992. Springer-Verlag.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[HR01a] K. Havelund and G. Roşu. Java pathexplorer — a runtime verification tool. In *Proceedings 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space, ISAIRAS'01*, Montreal, Canada, June 18–22 2001.

[HR01b] Klaus Havelund and Grigore Roşu. Monitoring programs using rewriting. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 135, Washington, DC, USA, 2001. IEEE Computer Society.

[HR01c] Klaus Havelund and Grigore Roşu. Testing linear temporal logic formulae on finite execution traces. Technical report, RIACS, 2001.

[HR02] Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 342–356, 2002.

[HR04] Klaus Havelund and Grigore Roşu. Efficient monitoring of safety properties. *Int. J. Softw. Tools Technol. Transf.*, 6(2):158–173, 2004.

[JM07] Saeed Jalili and Mehdi MirzaAghaei. Rverl: Run-time verification of real-time and reactive programs using event-based real-time logic approach. In *SERA '07: Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2007)*, pages 550–557, Washington, DC, USA, 2007. IEEE Computer Society.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[KMMP93] Yonit Kesten, Zohar Manna, Hugh McGuire, and Amir Pnueli. A decision algorithm for full propositional temporal logic. In *Computer Aided Verification*, pages 97–109, 1993.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[Lam80] Leslie Lamport. "sometime" is sometimes "not never" - on the temporal logic of programs. In *POPL*, pages 174–185, 1980.

[LBAK$^+$98] I. Lee, H. Ben-Abdallah, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. A monitoring and checking framework for run-time correctness assurance. In *Korea-U.S. Technical Conference on Strategic Technologies*, 1998.

[LKK$^+$99] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[LS07] Martin Leucker and Christian Scallhart. Monitor-based runtime reflection. In *FLACOS'07*, 2007.

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[MM84] Ben Maszkowski and Zohar Manna. Reasoning in interval temporal logic. In *Proceedings of the Carnegie Mellon Workshop on Logic of Programs*, pages 371–382, London, UK, 1984. Springer-Verlag.

[Mos86] B. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, Cambridge, 1986.

[MR03] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, (46):219–254, 2003.

[OGRG07] Omar Ochoa, Irbis Gallegos, Steve Roach, and Ann Gates. Towards a tool for generating aspects from medl and pedl specifications for runtime verification. In *Proceedings of the 7th International Workshop on Runtime Verification (RV)*, 2007.

[Pan01] P.K. Pandya. Specifying and deciding qauntified discrete-time duration calculus formulae using dcvalid. In *Proc. Real-Time Tools, RTTOOLS'2001 (affiliated with CONCUR 2001)*, Aalborg, August 2001. (Technical Report TCS-00-PKP-1, Tata Institute of Fundamental Research, Mumbai, 2000).

[Pan02] Paritosh K. Pandya. Interval duration logic: Expressiveness and decidability. *Electr. Notes Theor. Comput. Sci.*, 65(6), 2002.

[Ray96] Pascal Raymond. Recognizing regular expressions by means of dataflow networks. In *ICALP '96: Proceedings of the 23rd International Colloquium on Automata, Languages and Programming*, pages 336–347, London, UK, 1996. Springer-Verlag.

[RB06] Grigore Roşu and Saddek Bensalem. Allen linear (interval) temporal logic –translation to ltl and monitor synthesis–. In *Proceedings of 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2006.

[RH01] Grigore Roşu and Klaus Havelund. Synthesizing dynamic programming algorithms from linear temporal logic formulae. Technical report, RIACS, 2001.

[RH05] Grigore Roşu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engg.*, 12(2):151–197, 2005.

[SB06] Volker Stolz and Eric Bodden. Temporal assertions using aspectj. *Electronic Notes in Theoretical Computer Science*, 144(4):109–124, 2006.

[SC82] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. In *STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 159–168, New York, NY, USA, 1982. ACM Press.

[SS03] Usa Sammapun and Oleg Sokolsky. Regular expressions for runtime verification. In *Proceedings of the 1st International Workshop on Automated Technology for Ver ification and Analysis (ATVA'03)*, Taipei, Taiwan, December 10-12 2003.

[STY03] J. Sifakis, S. Tripakis, and S. Yovine. Building models of real-time systems from application software. *Proceedings of the IEEE*, 91:100–111, 2003.

[Tha00] H. Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems.* PhD thesis, Royal Institute of Technology, KTH, Mechatronics Laboratory, TRITA-MMK 2000:16, Sweden, 2000.

[UES00] Úlfar Erlingsson and Fred B. Schneider. Sasi enforcement of security policies: a retrospective. In *NSPW '99: Proceedings of the 1999 workshop on New security paradigms*, pages 87–95, New York, NY, USA, 2000. ACM.

[UT02] Naoyasu Ubayashi and Tetsuo Tamai. Aspect-oriented programming with model checking. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 148–154, New York, NY, USA, 2002. ACM Press.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *The First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

[ZCA91] Z. ChaoChen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

[ZKTR07] Karen Zee, Viktor Kuncak, Michael Taylor, and Martin Rinard. Runtime checking for program verification. In *RV'07*, 2007.