Runtime verification and monitoring of embedded systems

C. Watterson and D. Heffernan

Abstract: Ensuring the correctness of software applications is a difficult task. The area of runtime verification, which combines the approaches of formal verification and testing, offers a practical but limited solution that can help in finding many errors in software. Runtime verification relies upon tools for monitoring software execution. There are particular difficulties with regard to monitoring embedded systems. The concerns for arranging non-intrusive monitoring of embedded systems in a way that is suitable for use in runtime verification methods are considered here. A number of existing runtime verification tools are referenced, highlighting their requirement for monitoring solutions. Established and emerging approaches for the monitoring of software execution using execution monitors are reviewed, with an emphasis on the approaches that are best suited for use with embedded systems. A suggested solution for non-intrusive monitoring of embedded systems is presented. The conclusions summarise the possibilities for arranging non-intrusive monitoring of embedded systems, and the potential for runtime verification to utilise such monitoring approaches.

1 Introduction

To ensure that software applications operate satisfactorily in their final environment, extensive and costly efforts are involved during the software development cycle. Despite the evolution of improved software development practices, used to increase the correctness of the software design and implementation, the client users in the 'real world' fear failure of the system at some point. It is possible to persistently monitor software execution in its final environment, checking the observed behaviour and performance against specified rules. This approach of monitoring and checking, which can be termed runtime checking if the monitoring outputs are checked at runtime, can be used for various purposes. Potential applications of runtime checking include testing, debugging, verification, logging of errors, attempts at real-time fault recovery of the system to a safe state and maintenance/diagnosis procedures in the field. The focus of this paper is on the consideration of runtime checking mechanisms that are suitable for runtime verification of software in embedded systems.

Throughout this paper, the system in which monitoring, runtime checking, or runtime verification is performed is referred to as the 'target system', and the software application whose execution is being monitored is referred to as the 'target application'. The term 'monitoring' in this paper is used to refer to a specific monitoring task, the monitoring of software execution. Although the term monitoring can be used synonymously with observing, here monitoring is used to mean the combined task of observing and making use of the observations to 'keep track of' the target's behaviour. Observing the target software's execution is by definition practised at runtime, but the results of the observations can be examined later offline. This is known as offline monitoring; whereas when observed data are examined at runtime, the task is referred to as online or runtime monitoring. Such an approach is also referred to simply as monitoring; the term 'monitoring' will refer to runtime monitoring of software execution unless otherwise qualified.

Embedded systems can present particular challenges for monitoring. System internals are not easily observable, as many device features are incorporated deep within complex chip packages. This hinders the observation of software execution in such devices. Many embedded systems have limited resources or have real-time software requirements that must adhere to strict execution deadlines. These traits result in a need to minimise the overhead of monitoring; high overhead could compromise core system resources or affect scheduling, causing interference to the target application. The two concerns examined in this paper are: (1) the monitoring mechanism be capable of sufficiently observing the internal operation of the target system and (2) the overhead of the monitoring scheme be minimised so as to avoid interference with the normal behaviour of the target system.

The issue of what must be observed is a concern related to (1); this is considered here in the context of runtime verification. Runtime verification, detailed by Havelund and Goldberg [1], is a method of checking the correctness of programs and is described by Havelund and Roşu [2] and Lee *et al.* [3] as bridging the gap between formal verification and testing. Runtime verification is combined with test case generation by Artho *et al.* [4]. A survey of work that various groups have undertaken in the area of runtime verification details some existing runtime verification tools that employ methods based on various temporal logics and specification languages. The manner in which some of these runtime verification tools interface with the target system is also considered. Some conclusions

© The Institution of Engineering and Technology 2007
doi:10.1049/iet-sen:20060076

Paper first received 13th December 2006 and in revised form 3rd July 2007
The authors are with Centre for Telecommunications Value-Chain Research (CTVR), Department of Electronic and Computer Engineering, University of Limerick, Limerick, Ireland

E-mail: conal.watterson@ul.ie

Authorized licensed use limited to: HOSEO UNIVERSITY. Downloaded on October 6, 2009 at 06:20 from IEEE Xplore. Restrictions apply.

concerning the feasibility of runtime verification of embedded systems are drawn, based on the potential for using existing and emerging monitoring approaches.

Published reviews of monitoring approaches have been presented in the past by Plattner and Nievergelt [5] and Delgado *et al.* [6]. The monitoring of embedded systems raises special concerns, and this review considers various monitoring approaches, including: the use of internal system signals with hardware probes (hardware monitoring), the addition of code to the target system's software in order to perform operations related to monitoring at certain points in the application's execution (software monitoring), the combination of both approaches (hybrid monitoring) and the use of on-chip probes (on-chip monitoring). Selected references highlight the problems and advantages that have been considered in the past relating to these approaches. Emphasis is placed on achieving non-intrusive (or minimally intrusive) monitoring in embedded systems.

2 Monitoring of embedded systems

Monitoring involves observing the execution behaviour or performance of a target application in order to gain an insight into the operation of the software. Improved software development techniques have not lead to perfect software, and so there remains a need to complement these techniques with tools that help diagnose the behaviour of real software implementations. The rationale for monitoring and the fundamental concepts in monitoring are detailed in introductions to the field by Nutt [7], Plattner and Nievergelt [5], Plattner [8], Delgado *et al.* [6] and Thane [9]. Examples of runtime checking systems are detailed by Gates *et al.* [10] and Mok and Liu [11]. Runtime verification has been presented by Havelund and Roşu [2], Drusinsky [12] and Lee *et al.* [3] among others. General debugging and 'replay debugging' schemes have been described by Tsai *et al.* [13] and Thane [9] for example, whereas performance monitoring is described by Svobodova [14] and Haban and Wybranietz [15]. The following sections will consider monitoring solutions in the context of observing the execution of software in embedded systems, with runtime verification as a potential use for the observations.

In order to observe a target system, elements of a monitoring system, termed probes, are attached to the system (or placed within it) in order to provide information about the system's internal operation. As shown in Fig. 1, the probes provide an intermediate output from the system in addition to its end output, allowing the system to be seen as more than a 'black box'. Such probes can be actual hardware probes that monitor internal system signals. Alternatively, some code to perform monitoring can be added to the target software in order to output information about the internal operation of the program. This is termed instrumentation, and the added code can be regarded as 'software probes'.

There are two main concerns in arranging a probing mechanism that is suitable for runtime checking of embedded systems. First, the probes must be capable of observing enough information about the internal operation of the system to fulfil the purpose of the monitoring. Secondly, in adding such probes to the target system, its behaviour should not be affected. With regard to the first issue, the observations required are dependant on the purpose of the monitoring. The authors believe that a versatile runtime checking mechanism can be adapted for a wide variety of other monitoring purposes. The goal is therefore

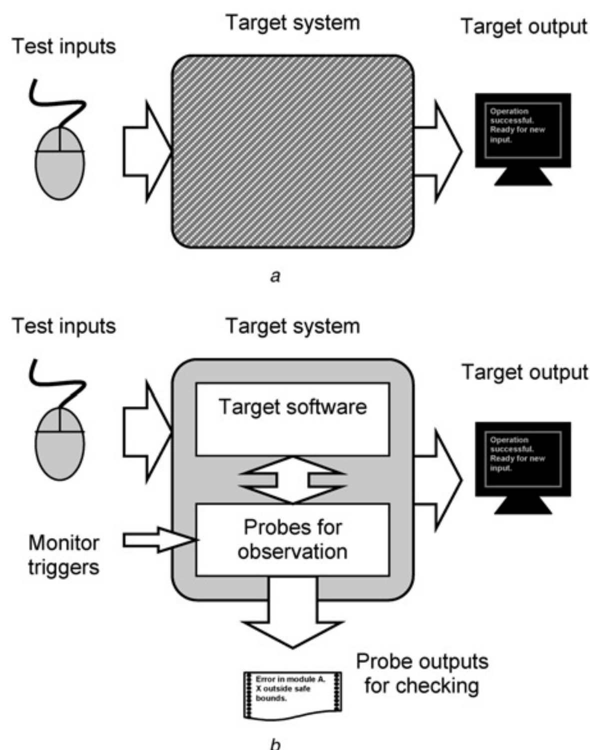

Fig. 1 *Target system*

a As a black box
b With outputs for monitoring

to support runtime checking using a new mechanism for monitoring software execution in embedded systems; experimental work will test the feasibility of such a mechanism for existing runtime verification approaches. Existing runtime verification approaches are examined in the following section, and some conclusions are drawn on the feasibility of supporting runtime verification of embedded systems. The nature of embedded systems hinders simple probing of internal hardware mechanisms. Paradoxically, the constraints on resource usage hinder the use of software probes that have insight into the internal operation of the system; such probes usually have high overhead that is likely to lead to interference with the normal operation of the target system. The importance of non-intrusive monitors and the problems with intrusive software monitoring systems are discussed by Harelick and Stoyen [16], where they present a less intrusive software monitoring system. Fryer [17] examines non-intrusive and 'low' intrusive monitoring in the context of more complex systems, and in particular embedded systems. Arranging a non-interfering probing mechanism that is capable of observing enough information about the internal operation of the system is a focus of later sections of this paper.

3 Observations necessary for runtime verification

It has been suggested by Nutt [7] in a tutorial on 'computer system monitors' that the most important questions to be answered before attempting to monitor a system are 'what to measure' and 'why the measurement should be taken'. The latter question essentially asks 'what is the general purpose of the monitoring'; is it debugging, testing or verification and so on? This in turn may dictate the answer to the first question; what exactly is to be observed? Part of our research will investigate the feasibility of runtime

Authorized licensed use limited to: HOSEO UNIVERSITY. Downloaded on October 6, 2009 at 06:20 from IEEE Xplore. Restrictions apply.

verification of embedded systems using a non-intrusive runtime checking mechanism.

With runtime verification, detailed observations of the target system execution behaviour are checked at runtime against properties that specify the intended system behaviour. These properties are often derived from the target application's software requirement specification, or indeed the properties to be monitored may form the entire specification. In either case, the properties are expressed formally, for example, using linear-time temporal logic (LTL) [18] to formally state properties or models of target applications for monitoring and verification. LTL is a branch of modal logic that considers the notion of time and order. LTL is similar to propositional logic but includes four temporal operators that can consider the values of propositions in the future or the past. These allow conditions such as 'sometime in the future this must hold true' and 'always in the past this must hold true'.

A means must also be provided to automatically link low-level observations of program execution behaviour, which can be emitted as events from the suitably instrumented target system, to the relevant monitored properties. The Monitoring and Checking (MaC) framework described by Lee *et al.* [3], and by Kim *et al.* [19] uses two languages: MEDL (Meta Event Definition Language) for specifications and PEDL (Primitive Event Definition Language) to map program level events to specification level events.

3.1 Expressing monitored properties using LTL

Havelund and Roşu [20] present a runtime verification tool named Pathexplorer (PAX) that uses LTL. Rules expressing intended program behaviour based on program actions at runtime (e.g. values assigned to variables) are monitored and verified by PAX. A synthesis algorithm is used to generate a verification algorithm from finite trace past-time LTL formulae. This verification algorithm checks that a finite sequence of events satisfies the formulae. The PAX architecture relies on instrumentation of the target application (using a tool named JTrek [21]) with monitoring code that emits events (changes in execution behaviour), which are dispatched to individual execution threads (that can execute on separate hardware) that check particular rules.

Java PAX, a related runtime verification tool, makes use of Maude's rewriting logic (Clavel *et al.* [22]) and provides future- and past-time linear temporal logics as detailed by Havelund and Roşu [23] [2] [24]. Specifications are used to instrument target applications with code to emit events, and also to generate observers. These observers are used in monitoring software execution to allow error-pattern analysis (checking for programming errors independent of specifications), as well as checking against the specification formulae. The Eraser algorithm of Savage *et al.* [25] has been implemented within the Java PaX framework to provide data race error-pattern analysis.

Havelund and Roşu [24] have concluded that LTL alone may not be the most appropriate formalism for logic-based monitoring in Java PAX. LTL has however been extended for more specific purposes. Metric Temporal Logic (MTL) as used in Temporal Rover by Drusinsky [12] allows the application of real-time constraints to the LTL operators, specifying duration bounds. Temporal Rover monitors program execution with respect to rules expressed using LTL and MTL, and runtime verification is performed using executable alternating automata. An extended LTL/MTL logic combined with times series constraints (LTLD) is also described by Drusinsky [26], for use with Temporal Rover and a remote version, DB Rover. LTLD allows the representation of properties such as stability, monotonicity, temporal min/max and temporal average values over time. To specify properties for monitoring, MaC by Lee *et al.* [3] uses a type of temporal logic that according to Havelund and Roşu [20] can be classed as a form of interval logic. This has been expanded by Sammapun *et al.* [27] to allow the use of regular expressions in event definitions, allowing simultaneous events to be more easily described.

The concept that different specific logic formalisms may be more appropriate than others, depending on the application, has lead to the approach of EAGLE, described by Barringer *et al.* in [28]. In addition to allowing the use of various logics, EAGLE allows new logics to be defined for use in runtime verification. These logics can be used in executable rules written as Java code. HAWK, introduced by d'Amorim and Havelund [29], is a temporal logic that extends the EAGLE framework. EAGLE requires the user to create a projection of the actual program state that is being monitored. In contrast, using HAWK, one only needs to refer, using specification formulae, to low-level events emitted by the instrumented target application.

3.2 Methods of observation to facilitate runtime verification

In order to observe the execution behaviour in enough detail to facilitate the checking of (possibly complex) properties expressed in formal logic, most runtime verification tools rely primarily on the addition of extra monitoring software to the target application. The Temporal Rover system of Drusinsky [12] relies on assertions placed as comments in the target application code. A code to verify/monitor the target at execution time is generated and compiled as part of the target application. The PAX architecture of Havelund and Roşu [20] also relies on instrumentation of the target application code. In the case of PAX, the additional code is used to emit events when predicates across program execution behaviour (e.g. values of variables) are updated. These events are received by a separate monitoring process that dispatches relevant events to separate verification processes.

Some runtime verification solutions make use of separate software components, some of which can be run on separate hardware (additional to the target system). Havelund and Roşu [2] mention that the 'observer' component of Java PAX can be located on a separate computer from that of the target application; events are transmitted from the target system over a socket. The MaC framework described by Lee *et al.* [3] adopts a similar 'event dispatch' approach, also relying on instrumentation of the target application in order to dispatch event information to an observer. This observer recognises events, and interfaces with a separate runtime checker component that performs the actual verification task.

3.3 Suitable runtime verification approaches for embedded systems

There are certain runtime verification approaches that are more suitable for adaptation for use in an embedded environment, although the feasibility of this will be the subject of further research by the authors. Certainly a more modular runtime verification approach is more suitable, as the verification engine and elements such as 'event recognition' (as in MaC) can be arranged on separate monitoring systems external to the target system. This

Authorized licensed use limited to: HOSEO UNIVERSITY. Downloaded on October 6, 2009 at 06:20 from IEEE Xplore. Restrictions apply.

means that the additional overhead incurred by the target system because of execution of a monitoring code can be reduced. Runtime verification can be scaled as noted by Havelund and Roşu [20]; a particular critical set of properties could be monitored, rather than a set of properties forming a complete specification of intended system behaviour. Approaches such as those used by PAX and MaC are suitable for this; selected rules rather than a comprehensive set can be monitored, thus reducing the monitoring overhead on the target system.

In summary, arranging a probing mechanism that emits observed execution behaviour as events should facilitate at least some limited form of runtime verification. The MaC framework is modular, which may allow experimentation to concentrate on different probing mechanisms to gather the necessary observations from an embedded system and relay them to the external runtime verification apparatus. Also, selected properties to be monitored can be simply expressed as a versatile mechanism for mapping language-specific low-level events to the high-level properties exists. The two main concerns for the remainder of this paper are achieving the non-intrusive probing of the target system and overcoming the low visibility of the internal operation of the system to an external monitoring component.

4 Approaches to monitoring

The classification of monitoring systems according to the three approaches used for probes, hardware, software and hybrid, has been used both in early surveys on monitoring such as that of Nutt [7] and later surveys such as that by Schroeder [30]. In the presentation of specific monitoring solutions, the categorisation continues to be used, for example, by Thane *et al.* [31]. These three approaches to monitoring, as well as on-chip monitoring, satisfy to different degrees the two requirements important for monitoring embedded systems; low overhead to avoid interference and sufficient observation of complex hardware.

4.1 Hardware monitors

Pure hardware monitors have existed for some time, as they include simple monitors that perhaps simply probe one or more internal signals of the target system hardware. As far back as 1975, Nutt [7] makes reference, in his survey on monitoring, to a monitoring system that monitors the signal sent to a processor's activity light on an IBM System/360 mainframe. More elaborate hardware monitors have been used in the decades since. A later example is that of Tsai *et al.* [13], where although the monitoring system is essentially a bus monitor (monitoring the internal system buses of the target), it consists of a substantial amount of additional hardware, including a separate microprocessor. A typical hardware monitoring arrangement or 'bus monitor' is shown in Fig. 2. As depicted, dedicated monitoring hardware is attached to the target system (i.e. to observe information sent over the internal system bus); the observed data is sent by the monitoring hardware for verification, external to the target system.

The non-intrusion benefits gained by using additional hardware for the monitoring function were recognised from quite early on in the area of performance evaluation of mainframe computers, with the non-intrusion feature being highlighted as a key advantage for hardware monitors by Calingaert [32], Karush [33], and Lucas [34]. Later work, such as that considered by Tsai *et al.* [13], also focused on the potential for minimising intrusion

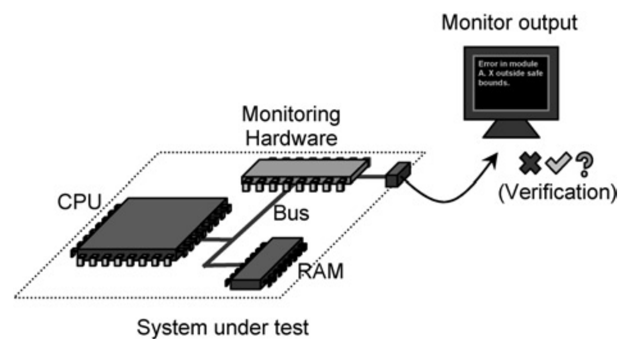

Fig. 2 *Simplified view of a typical hardware monitor*

caused by the execution of monitoring code by the target system, by using additional hardware for monitoring.

The difficulties of hardware monitoring have been emphasised for some time. In 1981 Gallo and Wilder [35] noted the problem of newer systems offering less physical probe points, and suggested at the time that hardware monitors were becoming obsolete. Haban and Wybranietz [15] also note the inapplicability of hardware monitors with respect to more complex systems. This view of traditional hardware monitoring approaches is repeated in later work suggesting alternative monitoring approaches. Calvez and Pasquier [36] suggest a hybrid monitoring approach and an on-chip monitoring approach is suggested by Shobaki and Lindh [37].

With the development of system-on-chip (SoC) devices, on-chip caching and more complicated processing and memory architectures have further reduced the visibility of program execution information system to external parts of the monitoring system that are separate from the target system (i.e. connected to the target only by hardware probes).

4.2 Software monitors

With software monitoring, there are different ways that the target system can be modified through the addition of software for monitoring. Fig. 3 shows how code to perform observations on the target application execution can be added to the target application code itself (instrumentation), or can take the form of a modification to the operating system of the target system, or the monitor can be a separate process. The software monitoring approach has long been

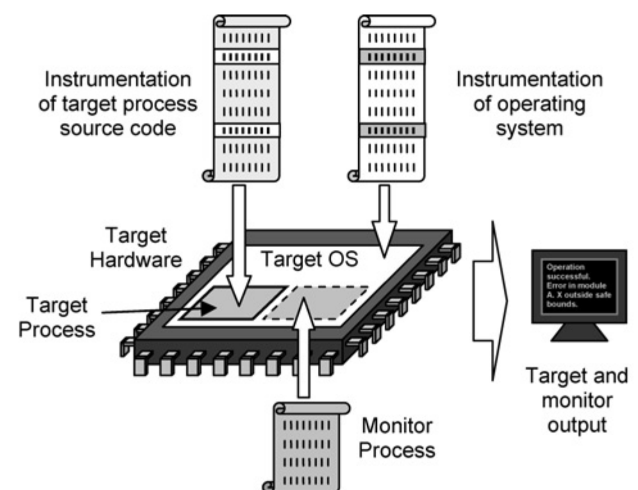

Fig. 3 *Abstract view showing possible locations of monitoring software within a target system*

Authorized licensed use limited to: HOSEO UNIVERSITY. Downloaded on October 6, 2009 at 06:20 from IEEE Xplore. Restrictions apply.

used; in 1975 Nutt [7] even suggests that it is an older approach than that of hardware monitoring. Older surveys by Calingaert [32] and Karush [33] on the subject of performance evaluation of mainframe computers make reference to running extra monitoring processes to trace execution steps, thus building a history of process execution behaviour for the system (job accounting).

One of the main drawbacks of software monitoring, as noted by Nutt, Calingaert and Karush, is the potential side effect from having the target system execute the additional software. Interference with the target system's normal operation may arise if the execution of the target software is delayed because of the execution of the monitoring code. However, it has been suggested by Lucas [34] even in examining older software monitoring tools, such as that of Stanley [38], that the effect of software monitoring can be minimised. Indeed software monitoring has been used for real-time and distributed systems as noted by Tokuda *et al.* [39], where tolerance of intrusion is particularly low. Tokuda *et al.* suggest an approach of building the monitoring apparatus permanently into the target system (also suggested by Svobodova [40]). The concept is to design a system with software probes permanently included, adjusting scheduling for the overhead of monitoring and allocating greater resources for the system as a whole. However, such an approach is not considered here by the authors, because even if such changes to the design process are possible, there are significant drawbacks in that more resources are required by the target system and overall system performance is impaired. In the area of embedded systems, where designs need to be as cost-effective as possible, the option of leaving software probes in the system is not particularly attractive, unless there are other cost/benefit considerations such as high-reliability requirements, where a cost premium can be justified.

The advantage of monitoring a system from a software monitor within the target is recognised by Deniston [41], in that such monitors have access to extensive information about the operation of a complex system, in contrast to the limited information available externally to hardware probes. Indeed, this flexibility is discussed by Rota and de Almeida [42] in more recent runtime monitoring research, where they explain their choice of the software monitoring approach, as compared to hardware or hybrid monitoring.

Applications of monitoring for runtime verification by Havelund and Roşu [2], Drusinsky [12] and Lee *et al.* [3] have mostly used the software monitoring approach. Indeed, almost all of the tools catalogued by Delgado *et al.* [6] use software monitoring.

4.3 Hybrid monitors

In general, hybrid monitoring refers to approaches that use a combination of additional software and hardware to monitor a target system, relying on the advantages of each approach and at the same time attempting to mitigate their disadvantages. By relying on aspects of hardware monitoring, such as making observations using physical interfaces to the target system, hybrid monitoring is less likely to interfere with the behaviour of the target system. Since more complex systems do not facilitate sufficient observation of execution behaviour using such physical probes, hybrid monitoring also relies on an additional monitoring code that is executed by the target system. Although the overhead of executing this code can potentially affect the behaviour and/or performance of the target system, the effects can be lessened compared to software monitoring because of the use of physical probes (i.e. a less monitoring code is required to sufficiently observe the system).

Hybrid monitoring was identified as distinct from software or hardware monitoring by Svobodova [14], although earlier monitors (particularly those relying on monitoring hardware) probably can be categorised as hybrid monitors. An abstract view of a hybrid monitoring arrangement is depicted in Fig. 4. In the arrangement shown, the source code of the target process (i.e. the process being monitored) is instrumented; an additional code is added to it to emit events when process features (e.g. variable values) being monitored are updated. These events are sent to the dedicated monitoring hardware, which analyses the events and checks them against the provided monitoring rules. The event analyser provides the dedicated monitor output.

Various hybrid monitoring systems have been developed; for distributed systems by Haban and Wybranietz [15] and Hofmann *et al.* [43], for real-time systems by Harelick and Stoyen [16] and for embedded systems by Calvez and Pasquier [36]. Hybrid monitoring has also been used for performance monitoring as discussed by Shobaki *et al.* [44].

4.4 On-chip monitors

The decreasing visibility of system operation to external observation affects not only traditional hardware monitors, but also hinders hybrid monitoring. On-chip monitoring, as described in the monitoring solution proposed by El Shobaki [44], increases the visibility of the target system operation for external parts of a monitoring system. Non-intrusive monitoring is facilitated through the use of additional on-chip hardware; the overhead of communicating observations to external checking mechanisms can be placed upon a small amount of dedicated hardware within the target system. This can be preferable to impacting on CPU time in a tightly scheduled real-time environment. The integration of on-chip tools is the subject of work by Walters *et al.* [45].

In addition to built-in, on-chip debugging capabilities incorporated at design time to support observation of internal system behaviour, there is a need for hardware interfaces allowing the observations to be communicated to external monitors. Some developments in this area are detailed by MacNamee and Heffernan in [46]; for example, the Nexus 5001 standard specifies a common debug interface for embedded systems. The widespread use of the IEEE1149.1 interface (JTAG) for various on-chip debuggers is also examined. MacNamee and Heffernan examine in [47] the possible uses of built-in on-chip debugging interfaces for requirements-based monitors.

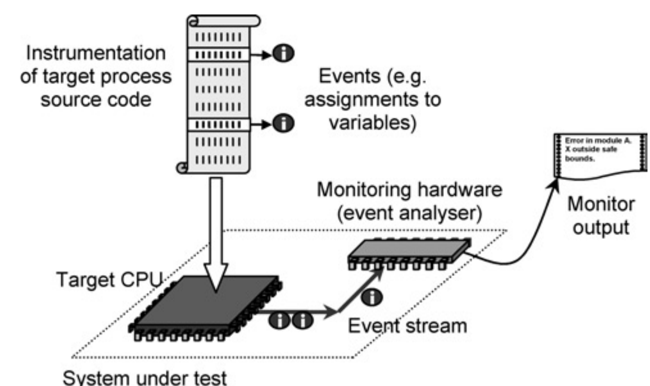

Fig. 4 *Abstract view of a hybrid monitoring arrangement*

Authorized licensed use limited to: HOSEO UNIVERSITY. Downloaded on October 6, 2009 at 06:20 from IEEE Xplore. Restrictions apply.

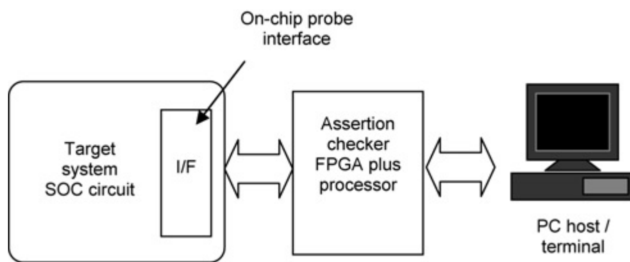

Fig. 5 *Assertion-based runtime debugger*

The use of field programmable logic arrays (FPGA) devices and customisable SoC designs offers new possibilities for hardware facilities supporting monitoring systems to be included at the design stage. The possibility of having an on-chip hardware based monitor is examined in such a context by MacNamee and Heffernan [47]. Drechsler [48] describes a method of synthesising on-chip hardware to check specification properties. In the context of monitoring a target system for verification, an approach has been proposed by El Shobaki and Lindh [37] of including a monitoring IP block on SoC devices to support runtime monitoring.

Verification approaches at a circuit design level (this is a level below the consideration of execution behaviour) can also be regarded as relevant to the monitoring of software execution in embedded systems, as circuit level properties have even more stringent requirements for monitoring, such as interface bandwidth, probing, complexity, speed, and so on. As complex hardware circuits, such as SoC devices, grow in complexity, the need for automated design verification solutions is becoming more apparent.

Although assertion-based verification methods have been used in software engineering for some time, the concept has more recently been applied to hardware design verification. The emergence of assertion languages, such as property specification language (PSL) [49], has accelerated the interest in assertion-based verification solutions for hardware-based systems. PSL is the IEEE Property Specification Language, which is defined in the IEEE 1850-2005 standard [49]. PSL provides a formal notation for the specification of electronic circuit behaviour, compatible with popular design languages, such as VHDL, Verilog, SystemC and SystemVerilog. Thus, an integrated specification and verification flow can be realised for multi-language hardware-based circuit designs.

Currently, there are design tools available to support assertion-based verification during the verification and simulation stages of the SoC design process. Such solutions use off-line analysis of trace debug data, based on assertion-based methods. However, a runtime verification solution can be beneficial for real-time verification. One of the first approaches to such a runtime verification solution at the SoC level is presented by Peterson and Savaria [50], where they describe a debug environment for complex hardware systems, for the real-time verification of prototype systems, which are running at full clock speeds. Peterson *et al.* refer to their solution as an 'assertion-based runtime debugger' (ABRD). A minimally invasive hardware on-chip probe interface collects relevant trace data and transfers this to an external board that incorporates an FPGA based assertion checker, which uses a real-time assertion-based verification approach to validate behaviour. For a given product design, ABRD files are generated with the aid of a special compiler that includes the specified assertions, expressed in an assertion language such as PSL. The properties, which are expressed in the register transfer logic level code at the product design time, can

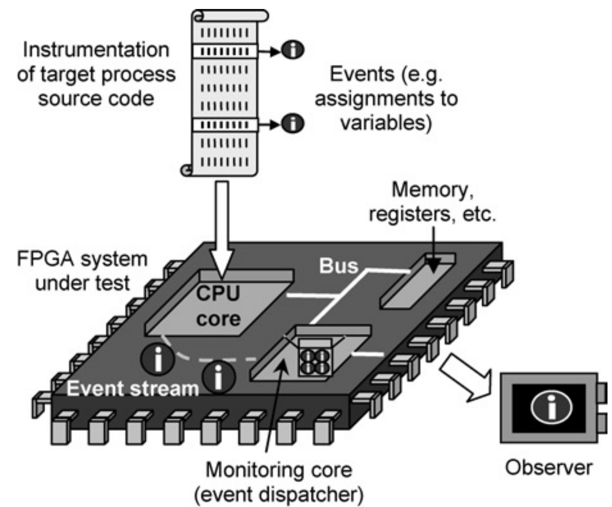

Fig. 6 *Theoretical on-chip (hybrid) monitoring arrangement*

be used in the on-line debug process. Fig. 5 shows the scheme.

The limited bandwidth of the probe interface and the loading effects at the SoC device interface are reported by the developers to highlight some limitations to the solution. However, the concept of extracting trace data in real time and applying assertion-based verification is a very important area of development for SoC hardware-level runtime verification. Emerging higher-density FPGA devices will more easily accommodate the on-chip interface, along with the assertion checker on the SoC chip, and thus reduce some of the reported limitations. These concepts are discussed by Fryer [51] in an embedded software environment where bandwidth and probing are less restricted than the SoC devices considered by Peterson and Savaria.

A theoretical hybrid monitoring arrangement that uses an on-chip monitoring core is shown in Fig. 6. In this arrangement, the target process may be instrumented to emit events (e.g. assignments to variables). This monitoring code does not need to include complicated communication code, as a dedicated monitoring core on the chip can instead perform the task of dispatching events from the chip to the external observer. External hardware observes the event information, and can perform whatever tasks are required (e.g. logging, verification, debugging).

5 Conclusions

Runtime verification offers a useful approach that can be used to check that software is correct in certain respects. It can be used to perform checking as an on-line safety measure, or as a testing tool for finding bugs. It also has the advantage that the approach has a strong formal basis. However, considering the difficulties faced in monitoring program execution, runtime verification tools will need further development for use in embedded and real-time systems if the intention is to evaluate the execution of the target application in its normal execution environment. Although arranging monitoring to solely use hardware probes would be advantageous with regard to non-intrusion, this is not really an option. The level of increasing hardware complexity today means that when relying on hardware monitors, there are problems in achieving the desired visibility of the target program execution. On the other hand, the need for non-intrusive observation of the target may preclude sole reliance on additional software.

Authorized licensed use limited to: HOSEO UNIVERSITY. Downloaded on October 6, 2009 at 06:20 from IEEE Xplore. Restrictions apply.

Nevertheless, some of the work in the area of runtime verification relies solely on the addition of extra software to the target application. At the very least, these approaches require a level of instrumentation that places enough overhead on the target system that the behaviour of software in an embedded system could be significantly affected. As evidenced by the modular nature of some runtime verification tools, the potential exists to optimise the use of additional hardware to perform the necessary processing of observations for verification. The combination of assertion-based verification with an on-chip monitor has been outlined by Peterson and Savaria in the proposal of [50]. If a minimally invasive scheme for probing an embedded device can be arranged by using on-chip and hybrid monitoring, it should be possible to provide the observations necessary for other runtime verification approaches.

Potential areas of investigation include incorporating on-chip hardware probes to observe execution behaviour, as well as utilising existing chip interfaces to provide the observations as events to an external monitoring system, such as a runtime verification engine. Rather than considering these areas solely in the context of execution monitors, it should be possible to use these monitoring approaches with existing runtime verification tools.

6 Acknowledgments

The authors wish to thank the Science Foundation of Ireland for its generous financial support for this research work through the Centre for Telecoms. Value-chain Research (CTVR). The authors also wish to express their gratitude to the anonymous reviewers whose comments have greatly improved the content of this paper.

7 References

1 Havelund, K., and Goldberg, A.: 'Verify your runs'. Proc. Verified Software: Theories, Tools, Experiments (VSTTE'05), Zurich, Switzerland, 10–13 October 2005

2 Havelund, K., and Roşu, G.: 'Monitoring Java programs with Java Pathexplorer', in Havelund, K, and Roşu, G. (Eds.), Proc. 1st Workshop on Runtime Verification (RV'2001) (13th Conf. Computer Aided Verification, CAV'01), Paris, France, 23 July 2001 Electronic Notes in Theoretical Computer Science (ENTCS), Elsevier vol 55, (2), pp. 200–217

3 Lee, I., Kannan, S., Kim, M., Sokolsky, O., and Viswanathan, M.: 'Runtime assurance based on formal specifications' in Arabnia, H.R. (Ed.): Proc. Int. Conf. Parallel and Distributed Processing Techniques and Applications (PDPTA 1997), Las Vegas, Nevada, USA, 30 June–3 July 1997, CSREA Press

4 Artho, C., Barringer, H., Goldberg, A., Havelund, K., Khurshid, S., Lowry, M., Pasareanu, C., Roşu, G., Sen, K., Visser, W., and Washington, R.: 'Combining test case generation and runtime verification', *Theor. Comput. Sci.*, 2005, **336**, (2–3), pp. 209–234

5 Plattner, B., and Nievergelt, J.: 'Monitoring program execution: a survey', *IEEE Comput.*, 1981, **14**, (1), pp. 76–93

6 Delgado, N., Gates, A.Q., and Roach, S.: 'A taxonomy and catalog of runtime software-fault monitoring tools', *IEEE Trans. Softw. Eng.*, 2004, **30**, (12), pp. 859–872

7 Nutt, G.J.: 'Tutorial: computer system monitors', *IEEE Comput.*, 1975, **8**, (11), pp. 51–61

8 Plattner, B.: 'Real-time execution monitoring', *IEEE Trans. Softw. Eng.*, 1984, **10**, (6), pp. 756–764

9 Thane, H.: 'Monitoring, testing and debugging distributed real-time systems', PhD thesis, Kungliga Tekniska Högskolan, Stockholm, Sweden, 2000

10 Gates, A.Q., Roach, S., Mondragon, O., and Delgado, N.: 'DynaMICs: comprehensive support for run-time monitoring' in Havelund, K., and Roşu, G. (Eds.): Proc. 1st Workshop on Runtime Verification (RV'2001) (13th Conf. Computer Aided Verification, CAV'01), Paris, France, 23 July 2001, Electronic Notes in Theoretical Computer Science (ENTCS), Elsevier, vol. 55, pp. 164–180

11 Mok, A.K., and Liu, G.: 'Efficient run-time monitoring of timing constraints'. Proc. 3rd IEEE Real-Time Technology and Applications Symp. (RTAS'97), Montréal, Canada, 9–11 June 1997, IEEE Computer Society Press, pp. 252–262

12 Drusinsky, D.: 'The temporal rover and the ATG rover' in Havelund, K., Penix, J., and Visser, W. (Eds.): Proc. 7th Int. SPIN Workshop on SPIN Model Checking and Software Verification, Stanford, California, USA, 30 August–1 September 2000, Lecture Notes in Computer Science (LNCS), Springer-Verlag, vol. 1885, pp. 323–330

13 Tsai, J.J.P., Fang, K.-Y., Chen, H.-Y., and Bi, Y.-D.: 'A noninterference monitoring and replay mechanism for real-time software testing and debugging', *IEEE Trans. Softw. Eng.*, 1990, **16**, (8), pp. 897–916

14 Svobodova, L.: 'Online system performance measurements with software and hybrid monitors', *ACM SIGOPS Oper. Syst. Rev.*, 1973, **7**, (4), pp. 45–53

15 Haban, D., and Wybranietz, D.: 'A hybrid monitor for behavior and performance analysis of distributed systems', *IEEE Trans. Softw. Eng.*, 1990, **16**, (2), pp. 197–211

16 Harelick, M., and Stoyen, A.: 'Concepts from deadline non-intrusive monitoring' in Frigeri, A.H., Halang, W.A., and Son, S.H. (Eds.): Proc. 24th IFAC/IFIP Workshop on Real-Time Programming (WRTP '99), Wadern, Germany, 30 May–3 June 1999, Elsevier

17 Fryer, R.: 'Low and non-intrusive software instrumentation: a survey of requirements and methods'. Proc. 17th AIAA/IEEE/SAE Digital Avionics Systems Conf. (DASC), Bellevue, Washington, USA, 31 October–7 November 1998, IEEE Press, vol. 1, pp. C22/1–C22/8

18 Pneuli, A.: 'The temporal logic of programs'. Proc. 18th IEEE Symp. Foundations of Computer Science (FOCS 1977), 1977, pp. 46–77

19 Kim, M., Viswanathan, M., Ben-Abdallah, H., Kannan, S., Lee, I., and Sokolsky, O.: 'Formally specified monitoring of temporal properties'. Proc. 11th Euromicro Conf. Real-Time Systems (Euromicro RTS'99), York, England, UK, 9–11 June 1999, IEEE Computer Society Press, pp. 114–122

20 Havelund, K., and Roşu, G.: 'Synthesizing monitors for safety properties' in Katoen, J.-P., and Stevens, P. (Eds.): Proc. 8th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002) (part of Joint European Conf. Theory and Practice of Software, ETAPS 2002), Grenoble, France, 8–12 April 2002, Lecture Notes in Computer Science (LNCS), Springer-Verlag, vol. 2280, pp. 342–356

21 JTrek, Digital Equipment Corporation (Compaq, HP), 1997

22 Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Quesada, J.F.: 'The maude system' in Narendran, P., and Rusinowitch, M. (Eds.): Proc. 10th Int. Conf. Rewriting Techniques and Applications (RTA-99), Trento, Italy, July 1999, Lecture Notes in Computer Science (LNCS), Springer-Verlag, vol. 1631, pp. 240–243

23 Havelund, K., and Roşu, G.: 'An overview of the runtime verification tool Java Pathexplorer', *Form. Methods Syst. Des.*, 2004, **24**, (2), pp. 189–215

24 Havelund, K., and Roşu, G.: 'Java Pathexplorer – a runtime verification tool'. 6th Int. Symp. on Artificial Intelligence, Robots and Automation in Space (i-SAIRAS'01), Montréal, Canada, 18–21 June 2001

25 Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T.: 'Eraser: a dynamic data race detector for multithreaded programs', *ACM Trans. Comput. Syst. (TOCS)*, 1997, **15**, (4), pp. 391–411

26 Drusinsky, D.: 'Monitoring temporal rules combined with time series' in Hunt, W.A. Jr. (Ed.): Proc. 15th Int. Conf. Computer-Aided Verification (CAV '03), Boulder, Colorado, USA, 8–12 July 2003, Lecture Notes in Computer Science (LNCS), Springer-Verlag, vol. 2725, pp. 114–117

27 Sammapun, U., Easwaran, A., Lee, I., and Sokolsky, O.: 'Simulation of simultaneous events in regular expressions for run-time verification' in Havelund, K., and Roşu, G. (Eds.): Proc. 4th Workshop on Runtime Verification (RV 2004) (in conj. w. 7th European Joint Conf. Theory and Practice of Software, ETAPS'04), Barcelona, Spain, 3 April 2004, Electronic Notes in Theoretical Computer Science (ENTCS), Elsevier, vol. 113, pp. 123–143

28 Barringer, H., Goldberg, A., Havelund, K., and Sen, K.: 'Rule-based runtime verification' in Steffen, B., and Levi, G. (Eds.): Proc. 5th Int. Conf. Verification, Model Checking and Abstract Interpretation (VMCAI'04), Venice, Italy, 11–13 January 2004, Lecture Notes in Computer Science (LNCS), Springer-Verlag, vol. 2937, pp. 44–57

29 D'Amorim, M., and Havelund, K.: 'Event-based runtime verification of java programs'. Proc. 3rd Int. Workshop on Dynamic Analysis (WODA 2005), St. Louis, Missouri, USA, 17 May 2005 ACM SIGSOFT Software Engineering Notes, ACM Press, vol. 30, pp. 1–7

30 Schroeder, B.A.: 'On-line monitoring: a tutorial', *IEEE Comput.*, 1995, **28**, (6), pp. 72–78

31 Thane, H., Sundmark, D., Huselius, J., and Petterson, A.: 'Replay debugging of real-time systems using time machines'. Proc. 17th Int. Parallel and Distributed Processing Symp. (IPDPS '03), Nice,

Authorized licensed use limited to: HOSEO UNIVERSITY. Downloaded on October 6, 2009 at 06:20 from IEEE Xplore. Restrictions apply.

France, 22–26 April 2003, IEEE Computer Society Press, pp. 288–295
32 Calingaert, P.: 'System performance evaluation: survey and appraisal', *Commun. ACM*, 1967, **10**, (1), pp. 12–18
33 Karush, A.D.: 'Two approaches for measuring the performance of time-sharing systems' in Denning, P.J., and Coffman, E.G., Jr. (Eds.): Proc. 2nd ACM Symp. Operating Systems Principles (SOSP'69), Princeton, New Jersey, USA, 20–22 October 1969, ACM Press, pp. 159–166
34 Lucas, H., Jr.: 'Performance evaluation and monitoring', *ACM Comput. Surv. (CSUR)*, 1971, **3**, (3), pp. 79–91
35 Gallo, A., and Wilder, R.P.: 'Performance measurement of data communications systems with emphasis on open system interconnections (OSI)'. Proc. 8th Annual Symp. Computer Architecture (ICSA'81), Minneapolis, Minnesota, USA, 12–14 May 1981, IEEE Computer Society Press, pp. 149–161
36 Calvez, J.P., and Pasquier, O.: 'Performance monitoring and assessment of embedded HW/SW systems', *Des. Autom. Embedded Syst.*, 1998, **3**, (1), pp. 5–22
37 El Shobaki, M., and Lindh, L.: 'A hardware and software monitor for high-level system-on-chip verification'. Proc. IEEE 2001 2nd Int. Symp. Quality Electronic Design (ISQED 2001), San Jose, California, USA, 26–28 March 2001, IEEE Computer Society Press, pp. 56–61
38 Stanley, W.I.: 'Measurement of system operational statistics', *IBM Syst. J.*, 1969, **8**, (4), pp. 299–308
39 Tokuda, H., Kotera, M., and Mercer, C.W.: 'A real-time monitor for a distributed real-time operating system', *ACM SIGPLAN Not.*, 1988, **24**, (1), pp. 68–77
40 Svobodova, L.: 'Performance monitoring in computer systems: a structured approach', *ACM SIGOPS Oper. Syst. Rev.*, 1981, **15**, (3), pp. 39–50
41 Deniston, W.R.: 'SIPE: A TSS/360 software measurement technique'. Proc. 24th Nat. Conf. ACM, 26–28 August 1969, ACM Press
42 Rota, S.R., and de Almeida, J.R., Jr.: 'Run-time monitoring for dependable systems: an approach and a case study' in Fraga, J.D.S. (Ed.): Proc. 23rd IEEE Int. Symp. Reliable Distributed Systems (SRDS 2004), Florianópolis, Brazil, 18–20 October 2004, IEEE Computer Society Press, pp. 41–49
43 Hofmann, R., Klar, R., Mohr, B., Quick, A., and Siegle, M.: 'Distributed performance monitoring: methods, tools, and applications', *IEEE Trans. Parallel Distrib. Syst.*, 1994, **5**, (6), pp. 585–598
44 El Shobaki, M.: 'On-chip monitoring of single- and multiprocessor hardware real-time operating systems'. Proc. 8th Int. Conf. Real-Time Computing Systems and Applications (RTCSA 2002), Tokyo, Japan, 18–20 March 2002
45 Walters, G., King, E., Kessinger, R., and Fryer, R.: 'Processor Design and Implementation for Real-Time Testing of Embedded Systems'. Proc. 17th AIAA/IEEE/SAE Digital Avionics Systems Conf. (DASC), Bellevue, Washington, USA, 31 October–7 November 1998, IEEE Press, vol. 1, B44/1–B44/8
46 MacNamee, C., and Heffernan, D.: 'Emerging on-chip debugging techniques for real-time embedded systems', *Comput. Control Eng. J.*, 2000, **11**, (6), pp. 295–303
47 MacNamee, C., and Heffernan, D.: 'Implementation approaches for requirements-based monitors for embedded systems'. Proc. IEEE IC Test Workshop (ICTW 2004), Limerick, Ireland, 13–14 September 2004, E&CE Dept, University of Limerick.
48 Drechsler, R.: 'Synthesizing checkers for on-line verification of system-on-chip designs'. Proc. 2003 IEEE Int. Symp. Circuits and Systems (ISCAS'03), Bangkok, Thailand, 25–28 May 2003, IEEE, vol. IV, pp. 748–751
49 IEEE 1850-2005: 'IEEE Standard for property specification language (PSL)', IEEE Standards Association, 2005
50 Peterson, K., and Savaria, Y.: 'Assertion-based on-line verification and debug environment for complex hardware systems'. Proc. 2004 IEEE Int. Symp. Circuits and Systems, Vancouver, British Columbia, Canada, 23–26 May 2004, IEEE, vol. II, pp. 685–688
51 Fryer, R.E.: 'FPGA based CPU instrumentation for hard real-time embedded system testing', *ACM SIGBED Rev*, 2005, **2**, (2), pp. 39–42

Authorized licensed use limited to: HOSEO UNIVERSITY. Downloaded on October 6, 2009 at 06:20 from IEEE Xplore. Restrictions apply.