

Automated Code Synthesis for Run-Time Verification of Distributed Embedded Systems*

István Majzik and Gergő Horányi

Dept. of Measurement and Information Systems

Budapest University of Technology and Economics, Budapest, Hungary

majzik@mit.bme.hu, gergo.horanyi@gmail.com

Abstract

This paper presents the design and implementation of a run-time verification approach for distributed embedded systems. We developed a technology for the integrated synthesis of the source code of the running applications and the source code of the monitors for run-time verification, based on formal models (timed automata) and formal property specifications (a variant of timed computation tree logic and Live Sequence Charts). The local monitors (focused on control flow checking) and the system-level monitors are able to detect random failures in the execution environment as well as the violation of the specified properties due to improper configurations or invalid design assumptions.

1 Introduction

Run-time verification is a technology that allows automated construction of run-time monitors to check formally specified system properties. It can be considered as a form of on-line testing that checks whether the specified properties (that might already been checked on the basis of design models in the development phase) are satisfied after deployment, in the run-time environment of a system. Run-time verification is able to detect when random failures occur in the execution environment (e.g., a hardware failure or a communication failure), when an improper configuration of the system causes that specified properties are violated, or when the context of the system execution does not satisfy the assumptions considered in the design phase.

Such kind of anomaly detection (considering hardware, communication and configuration failures) and assumption monitoring tasks are especially important in dependable and safety-critical computer systems. In dependable systems, fault detection mechanisms are the substantial facilities for developing fault tolerance, i.e., to check for erroneous states and then perform recovery using redundancy and diversity. In safety-critical systems, one of the basic principles for assuring safe behaviour is *reactive fail-safety*: proper detection and negation of hazardous faults that occur in a system component that implements a safety-related function. Here the detection is the responsibility of a separate component that shall be independent in order to avoid common-cause failures. This principle appears, for example, in IEC 61508 (the generic standard for safety-related electronic systems) and EN 50129 (the standard for electronic systems for railway signalling). In the design phase a strategy shall be defined to detect failures and to protect the system from the safety-related consequences of these failures. Among others, highly recommended techniques are *dynamic fault detection* (on-line dynamic testing to check the proper operation of the safety-related system) and *program sequence monitoring* (temporal or logical monitoring of the program sequence).

In this paper we describe a technology for the integrated synthesis of program source code and monitor code for run-time verification, based on formal models and formal property specification. To detect random operational faults, the monitor components support *control flow checking* (to detect anomalous

*This work was partially supported by the project BelAmL.H 2nd Phase and was also connected to the scientific program of the “Development of quality-oriented and harmonized R+D+I strategy and functional model at BME” project supported by the New Széchenyi Plan (TÁMOP-4.2.1/B-09/1/KMR-2010-0002).

control flow of the computation) and in limited cases some *data acceptance checking* (to detect anomalous values of intermediate or output data).

We apply our technology to distributed microcontroller systems in embedded control applications. Our target systems implement real-time, state-based, event-driven behaviour, which is characterized by a complex control flow, but less complex data processing. The behaviour can be modelled by formalisms (finite state machines and timed automata) that are relatively easy to map to program source code. The required properties are specified using invariants, scenarios, and temporal logic formulas expressing reachability of given states. The monitors are synthesized on the basis of these property specifications. The integrated synthesis of the program code and the monitor code offers an optimized information flow between the running programs and the monitor.

Our work was inspired by the classical code generation approaches that have been proposed to support the synthesis of monitors and executable assertions. An automaton-theoretic approach was presented in [6] based on implementing alternating automata. In [8] and [7] temporal logic checking is based on a formula rewriting engine. These approaches are typically not directly applicable in low-cost embedded systems where the processing power and memory constraints heavily restrict the applicable technology. In [17] a methodology is defined for developing C wrappers for real-time systems based on restricted linear temporal logic specifications. Among the first commercial tools, the Temporal Rover [5] is capable of translating temporal logic formulae into executable assertions for various programming languages, but the exact method of evaluating formulae in the generated code is not public. In [15] a code generation method was developed that converts linear temporal logic formulae into highly efficient assertion code that is applicable even in case of resource constraints and without restricting the expressive power of the temporal logic. We adapted these ideas to models and characteristics of distributed real-time systems.

The properties of our approach can be summarized as follows.

- We support model-based design, where the application is designed using a formal modelling, verification and validation environment. Distributed real-time systems are modelled as networks of timed automata, and required properties are specified (and verified) using a variant of timed computation tree logic and Live Sequence Charts [4]. The properties that were verified in design time (including basic checking of the formalization of these properties) are also monitored in run-time to detect run-time anomalies and violation of assumptions.
- We integrate the model based synthesis of program source code with the synthesis of the monitor code. The monitor uses the model as reference, while the program source code (that implements the behaviour specified in the model) is instrumented automatically to provide the run-time information that is required by the monitor for fault detection.
- We offer a hierarchical structure of monitors that is configurable in order to detect faults at the level of separate microcontrollers (local monitoring) and also at the level of the distributed application (system-level monitoring). The automated code generation offers optimized monitor code for a given configuration.
- The properties checked by the monitors are not restricted to invariant properties. The checking of some existential properties is useful in the testing phase when a restricted set of test runs (belonging to a given test suite) is monitored.

The paper is structured as follows. After a summary of the application domain and the automated source code generation technology (Sect 2) we introduce the concept of hierarchical monitoring (Sect. 3) based on formal property specification. The next two sections present the monitor synthesis technology

for local monitoring (Sect. 4) and system-level global monitoring (Sect. 5). Here we discuss the underlying property specification languages for monitor synthesis as well as the integration with program source code generation. Finally Sect. 6 concludes the paper and outlines the directions of future research.

2 Automated Source Code Generation Based on Formal Models

In our work we focus on state-based, event-driven applications that can be implemented using relatively simple microcontrollers connected to a limited set of sensors, actuators, and communication facilities including wireless communication. The target applications have real-time requirements that need considering timeliness in the computation. Our first application platform was the modular *mitmót* system developed by the Dept. of Measurement and Information Systems at the Budapest University of Technology and Economics [19]. The stackable *mitmót* modules [13] allow the construction of various devices from 8-bit or 32-bit processor, communication, sensor, and actuator modules. Reference applications include distributed sensor networks, mobile robot control and ambient intelligence applications.

2.1 Modelling and Verification

We found that the modelling formalism of networked *timed automata* (TA) [1] supported by the UPPAAL tool environment [3] fits to this type of applications. Roughly speaking, the formalism is based on finite state machines – with states and transitions – extended with data types (Boolean, bounded integers and arrays). Modelling timing aspects is possible using special clock variables. Time and data dependent behaviour can be modelled since state transitions are extended with guards (that can refer to the values of the variables) and actions (that perform value assignment, including the reset of clock variables). Moreover, states are characterized with time invariants. Distributed applications can be modelled constructing networks of timed automata with synchronous communication: sending and receiving of messages on channels are specific synchronization actions on state transitions.

The UPPAAL tool environment supports the design, verification and validation of TA models allowing model construction, simulation, and formal verification (model checking) of properties [20] [3]. The properties are specified using a subset of the timed branching time temporal logic (TCTL). It is based on the computation tree logic CTL [9], but the use of nested temporal operators is restricted. Moreover, in the recent version of UPPAAL, scenarios can be specified using a variant of Live Sequence Charts (LSC). The LSC formalism [4] defines extensions in comparison with traditional message sequence charts, for example it can express mandatory, forbidden and provisional behaviour. Scenarios that specify the interactions among automata can be checked by the UPPAAL verification engine.

2.2 Mapping the Models to Source Code

In our previous work [10] we elaborated a technique for automated source code generation for distributed embedded systems. In the next subsections we present this technique as the monitor synthesis will be integrated with it.

The synthesis of the source code of the application is based on the TA models. Considering our target applications (microcontroller based, loosely coupled distributed systems), in the models we restrict the use of shared variables and so-called committed states that would need system-wide synchronization.

Although the TA models are platform independent, we aimed at generating source code that can be compiled and executed on the given platform. To allow this, the code generator has the following responsibilities:

- It assembles source code snippets that correspond to the semantics of TA, namely implement the operational semantics based on the concept of states, transitions, guards and actions. The mapping of the behaviour of TA to the constructs of common structured programming languages (in our framework the C language) is relatively easy (see later in this section).
- Mapping of the semantics of advanced modelling concepts, namely the use of clock variables (timing) and synchronous communication, requires platform level services that are directly implemented on the given platform. Handling of clock variables is based on *timing services* that on our platform utilize the hardware timers of the processor module. The synchronous communication needs a middleware-like *communication services* that on our platform utilize the wireless radio module. These “semantic-oriented” core services are defined as abstract services for the source code generator, and their mapping to concrete functions on a given platform is defined in an external configuration file. (This way the code generator shall not be modified when the application is ported to another platform.) The platform services are thoroughly tested (according to precise coverage criteria) on the basis of the required semantics.
- Besides these core services (required to implement the semantics of the TA) other, application-specific platform services are also needed. On our platform they refer to the usage of sensors (inputs) and actuators (digital outputs). In this case, since these do not appear as specific model elements in the TA formalism, the meaning of some existing model elements shall be extended. For example, a specific variable of the model is identified as the variable belonging to a given digital output of the platform. The code generator, besides implementing the usual semantics of variable assignment, additionally calls a platform service that sends the value of the given variable to the output. The identification of the variable and the corresponding platform service are defined in an application-specific configuration file that is also processed by the code generator.

According to the above listed responsibilities, the code generator maps systematically each TA model to source code that can be compiled, linked with the platform services and uploaded to the microcontroller. The code generation process is summarized in Figure 1: the code generator reads and explores the concrete TA model, systematically maps each TA model element to a well-defined source code snippet, assembles these, and inserts the necessary calls to the platform level services.

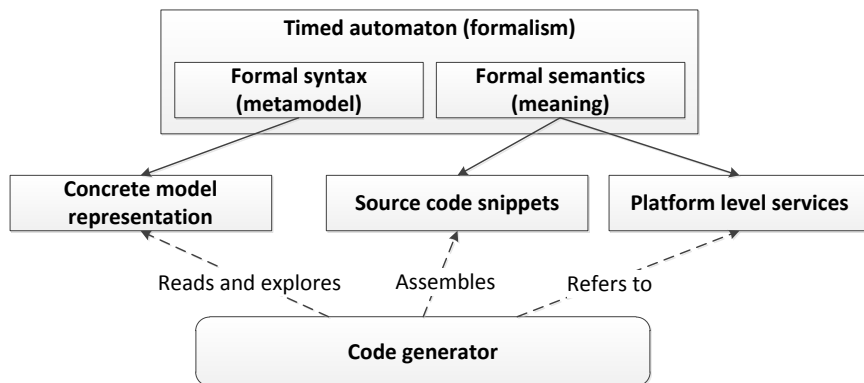


Figure 1: Overview of the code generation process

2.3 Template Based Code Generation

The source code snippets and the structure of their assembly in the generated code are defined in a code generation template, created with the Java Emitter Templates framework [11].

The basic template generates three functions for each state of the TA. There is one function for entering the state, which updates state variables and starts to listen on the communication channels defined on the outgoing transitions from the state (listening to a channel is a platform service). The waiting function (called from the state entering function) contains a loop that waits in the state for the enabledness of outgoing transitions. In this function, eventually an enabled transition is selected (non-deterministically, if there are multiple enabled transitions) and then the exit function of the state is called. It implements the exit from the state on the selected transition: it performs the variable assignments (*updates*) on the transition and sends the synchronization messages (if any). After that the new state designated by the transition is entered by calling its entering function – and the processing cycle starts again.

The template based generation of the source code allows an easy implementation of configurable extensions in the source code. We defined and implemented several extensions that are, however, restricted to complementary services that do not modify the core behaviour that corresponds to the semantics of the TA. This way it is assured that the behaviour of the program, considering the reachability of states and state transitions, does not deviate from the behaviour of the verified model.

Currently such extensions include (1) the insertion of executable assertions to perform self-checking, (2) the insertion of logging in states and transitions in order to support the computation of state and transition coverage metrics during testing. These extensions are implemented using application-specific platform services configured in the same way as described above, for example local (*memory-based*) or remote (*message-based*) logging can be configured without the modification of the code generator.

By extending the code generation template, the complementary services can be hooked onto model elements (states, transitions, variables etc.) and their execution is associated with the corresponding semantic concepts (entering a state, firing a transition, sending a message etc.). This way the model elements and concepts that are referred in the property specifications can also be addressed. This opens a way for supporting run-time verification of the behaviour.

3 Hierarchical Monitoring

As presented in Sect. 1, the goal of run-time monitoring is the detection of anomalies in the behaviour of the checked system. The reference information that is required to detect the anomalies is based on the following sources:

- The TA model describes the behaviour of the automata that are implemented in the microcontrollers. The syntactically allowed control flow is represented by the states and transitions, the communication (synchronization between automata) is modelled explicitly, and the state invariants are also given in the model.
- Safety and liveness properties are specified in the form of temporal reachability of given states and values of given variables using a subset of TCTL. Some TCTL expressions may refer to a single automaton (i.e., only its states and variables are included in the expression), while other expressions may refer to a (sub)network of automata (states and variables of multiple automata are included).
- A lightweight form of specifying system properties is the description of scenarios that characterize the behaviour (especially the interactions) of the network of automata. The UPPAAL tool envi-

ronment supports a restricted subset of the LSC formalism. Universal types of LSC are used in invariant mode, i.e., they specify a property the system should always satisfy. These restrictions, that are necessary to perform design time verification in the current version of UPPAAL, can be resolved in the run-time verification phase.

According to this set of reference information, the local checking of automata and the global (system-level) checking of the (sub)network of automata can be distinguished. This is motivated by the fact that the behaviour should be checked as fast as possible with minimal overhead. Local monitoring of a separate microcontroller (automaton) focusing on its control flow and local invariants offers a fine granular checking that results in short detection latency. Moreover, local monitors can be deployed close to the monitored microcontrollers, which reduces the communication overhead. However, system-level temporal logic criteria as well as interaction scenarios can be checked by system-level monitors that should receive information from all affected automata.

Accordingly, in our environment monitoring is implemented hierarchically considering the scope of the properties to be checked and the available run-time information.

- The responsibility of a local monitor is to check (1) the actual control flow of a microcontroller on the basis of its timed automaton model, (2) the timeliness of the behaviour on the basis of invariants given in the timed automaton model, and (3) the temporal correctness (including safety and liveness) of the local behaviour on the basis of TCTL properties that refer only to the given automaton.
- The responsibility of a system-level monitor (in short, global monitor) is to check (1) temporal correctness of the behaviour of interacting microcontrollers on the basis of TCTL properties that refer to several automata, and (2) communication of microcontrollers on the basis of LSC scenarios.

This way synthesis of local and global monitors is based on the formal model of the application (the timed automata model) and the formalized specification of the required properties (subset of TCTL, and the Live Sequence Chart variant). As presented in Sect. 2, the same formal model is used for the synthesis of the program code of the microcontrollers. In addition, the synthesis of the program code takes into account the needs of the monitoring as well: the program is extended to allow (1) access to local information about the states and variables to be observed by the local monitor, and (2) access to messages to be observed by the global monitor. This information transfer can be optimized since only those states/variables/messages are transferred that are involved in the specified properties and thus need to be observed.

The monitors are useful in the following use cases:

- Run-time verification: This is the use case motivated in Sect. 1. Transient and permanent operational faults, violation of assumptions (e.g., considering the underlying communication network) can be detected. The monitoring of certain properties can be seen as valuable evidence that might be considered in run-time certification [18]. Since run-time information is available in the form of a concrete execution trace that characterizes the running of the distributed application, checking is focused on invariant properties (control flow, data values, temporal logic properties with universal path quantifier, and universal LSC that refer to each trace).
- Design-time verification during testing: The monitors form (part of) the test oracle that decides whether the behaviour is acceptable considering the execution of a given test suite. Since in this case not only a single trace but a set of traces can be processed (corresponding to the execution of each test case), checking can also include existential properties (branching time temporal logic

properties with existential path quantifier and existential LSC that specify sample runs of the system). As testing is performed on the real system, design and implementation faults related to the platform services can also be detected.

Both use cases can be considered complementary to the formal verification (model checking) of the design models, since checking is performed during the execution of the application (either in usage or executing test cases) using the concrete platform and the related services. These are rarely modelled in detail (especially considering the potential failures) and are rarely involved in model checking due to complexity limitations.

4 Synthesis of Local Monitors

The main responsibility of our local monitors is control flow checking. It is motivated by the fact that as experience and fault injection experiments showed, a majority of transient faults in microprocessors result in control flow errors, i.e., the change of the normal (reference) flow of executed instructions. In the literature, low level approaches of control flow checking take the binary code of programs as reference and observe the actual flow of instructions on the external bus of the processor. Other approaches use the high-level source code of the program as reference, and rely on receiving explicit information (so-called signatures that identify the actual states) from the running program. This is especially useful in cases when the instruction fetch is not observable due to internal instruction cache in the microprocessor. Even such high-level approaches can reduce the percentage of undetected errors by 10 to 30%, depending on the built-in hardware-based error detection mechanisms and the structure of the code [14]. In our case the reference is the TA model of the program (that is also the basis of code generation), the program shall send explicit state messages (signatures) to the monitor, and the monitor checks whether the current sequence of signatures is allowed by the TA (abstracting from data dependencies).

According to this concept, the local monitor consists of two parts. There is a component that collects the signatures received from the monitored program and another component that evaluates each state signature on the basis of the internally stored TA model. This way the application-specific part of the monitor is only a data structure that is filled in on the basis of the actual TA model, storing the states and transitions (with unique identifiers) and their valid successors. The synthesis of the monitor is performed by using a specific JET template that explores the TA model, constructs the data structure and implements the signature evaluation.

The template for generating the application source code is extended as follows. Whenever a state is entered or a transition is selected to fire then a message is sent to the monitor using a configurable platform service. Each message contains the unique identifier of the state or transition exactly as stored in the reference. If the platform service (responsible for the communication with the monitor) is configured as a synchronous communication service then the application program can be stopped when an error is detected, otherwise the error signal is asynchronous. Note that the loss, duplication or bad ordering of signatures during communication is also checked by using sequence numbering.

Besides the classic control flow monitoring there are a few additional checks supported by the local monitor.

- In case of waiting for synchronization the application may suspend sending signatures. In order to distinguish this case from a program crash or other fail-silent failure condition, the application is responsible for sending “heartbeat” signals to the monitor. Accordingly, the application source code generation template is modified to define a timer and send heartbeat signals periodically.
- To check the real-time constraints of staying in certain states, the monitor is able to check the satisfaction of *time invariants* (state invariants in the TA model that contain clock variables). To

do this, the values of the corresponding clock variable is sent to the monitor when a state is left, and the reference data structure in the monitor contains the clock boundaries.

5 Synthesis of System-Level Monitors

While the local monitors focus on the detection of control flow related errors of separate applications (automata), the global monitors offer examining the behaviour of interactions and communication. It is motivated by the fact that programs with correct control flow may present problems at system level considering for example faults in the communication network or in the related platform services, or the incorrect assumptions in system level modelling. As envisaged in [18], at system level it is feasible to introduce checking based of formalized system properties, especially safety properties. We support the monitoring of two types of formalized properties: expressions in the TCTL variant, and scenarios in the LSC variant of UPPAAL. (Note that the checking of such TCTL expressions that refer only to a single automata can be deployed as a local monitor, however, the main use of TCTL expressions is to formalize system-level properties, this way we discuss the synthesis of the related monitor here.)

5.1 Checking Temporal Logic Properties

Temporal logic languages were suggested by Pnueli [16] in 1977 for reasoning about concurrent programs. Since then several temporal logic variants were defined for checking and proving temporal correctness of software, communication protocols and hardware devices. The core concept of checking temporal criteria is to represent the checked system as a finite state-transition system and prove that specific propositions hold. In comparison with classical tableau-based, automaton-theoretic or symbolic model checking approaches, run-time verification shall focus on checking concrete execution traces (which may be finite). It is solved by constructing proper observers that evaluate given temporal logic formulae over execution traces, e.g., as described in [15]. Our monitor is based on observers that (1) are specialized to the TCTL variant of UPPAAL and (2) support the use of monitors also in system testing as described in Sect. 3.

Accordingly, we created observers for the temporal operators of UPPAAL TCTL: *EF* (possibly), *AG* (invariantly), *EG* (potentially always), *AF* (eventually), and $\rightarrow$ (leadsto, where p leadsto q is equal to $AG(p \text{ imply } AF q)$). Temporal operators can not be nested (they are allowed only at the beginning of expressions referring to Boolean combination of state properties), this way the observers are relatively intuitive.

Figure 2 shows the overall template of the observer belonging to the *AF* operator. An *AF p* expression, where p is a state property, is evaluated to true if and only if all possible transition sequences eventually reach a state satisfying p . In a run-time verification use case only safety properties (as opposed to liveness [18]) are checked, while in the system testing use case the path quantifiers are interpreted on the set of traces belonging to the execution of the test suite. This is represented in case of *AF* in Figure 2. It is also shown how a counter-example is stored.

The synthesis of the monitor is based on the property specifications to be checked, using the observer template corresponding to the temporal operator of a given property. Checking is based on explicit messages received from the program whenever a state referred in the state property is entered or when a variable contained in the property is assigned a new value. The values of the clock variables (timers) are sent at the exit of the states. If a message is received then the related observers are determined and these evaluate the message by performing a step.

The application source code generation template is extended to send the above mentioned information to the monitor using a configurable platform service. Note that the integrated synthesis approach

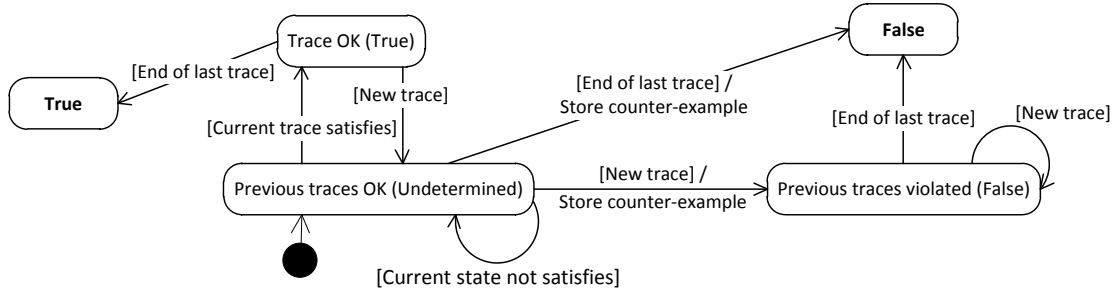


Figure 2: Observer belonging to the AF (eventually) temporal property

allows the reduction of the overhead as only those information is sent to the monitor that is needed to evaluate the current set of temporal logic properties.

An interesting extension is needed to check the special state property “deadlock” that is included in the TCTL variant of UPPAAL. It is evaluated to true by the model checker if the system reaches a deadlock. Our monitor checks deadlock as follows: Deadlock in the system is reported, if all the automata of the system entered or exited a local state where there is no enabled outgoing transition and there is no currently disabled transition that is only blocked by a time constraint and may be enabled later. This approach does not consider the possibility that a transition is blocked because it is unable to synchronize, this way such kind of deadlock is not reported by the monitor. However, false positive detection of deadlocks is avoided. The application source code generation pattern is extended to notify the monitor about the local states satisfying the above mentioned condition.

5.2 Monitoring scenario based requirements

The current development version of UPPAAL (version 4.1.4) has the ability to specify LSC scenarios, and the built-in verifier can check whether the system satisfies these scenarios or not. LSC scenarios, as they have formal semantics, make the formalization of system-level requirements easier (in some cases possible at all) and more intuitive than in case of the given TCTL variant. For example, a property like “if process B sends a message m_1 to process A , and C sends m_2 to D (in any order), then B must send m_3 to C within 1 to 3 time units” can be formalized [12].

An LSC is composed of a pre-chart and a main chart. If the behaviour described by the pre-chart is encountered, then the behaviour described by the main chart must be matched to satisfy the scenario. A chart contains a finite set of automaton instances (processes), each of them is represented by an instance line with a finite set of locations. The locations denote the possible points of communication and computation. Communication is achieved by messages between two instance lines. Locations may be associated with conditions and assignments (guards and variable updates in the TA). A condition can be hot, which means its satisfaction is mandatory, or can be cold, which means its satisfaction is possible. If a cold condition is evaluated to false then the chart is aborted. If a hot condition is violated then the chart is terminated, and the scenario is not satisfied (failed).

The synthesis of the monitors that check LSC scenarios is based on the algorithm presented in [12] and [2]. Since this algorithm addressed the needs of design-time verification of TA models, some adaptations and simplifications (detailed at the end of this section) were possible in our case of run-time verification. The steps of the monitor synthesis are summarized in Figure 3.

In case of a given LSC scenario we construct an observer automaton in the form of a TA, which allows using our source code generator (described in Sect. 2) directly to synthesize the code of the monitor (note that observers are more complex in this case than in the case of checking temporal logic properties, this

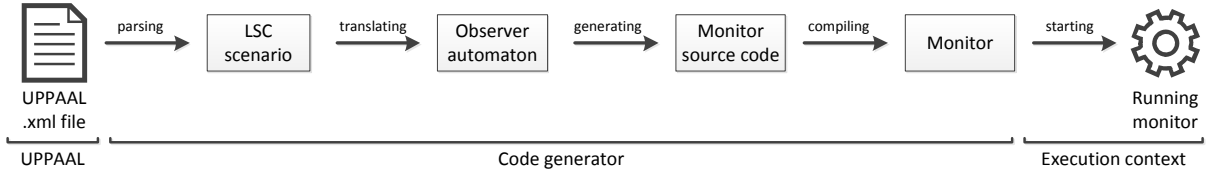


Figure 3: The steps of creating monitors checking scenario based requirements

way using the code generator becomes feasible). The generated code can be compiled without additional modifications.

The monitor checking a given scenario evaluates the following run-time information: (1) the current state of the automata involved in the scenario, (2) the value of the variables in the conditions of the scenario, and (3) the successful synchronizations performed by the automata involved in the scenario.

The application source code generation template is extended to send this information to the monitor in the form of messages, using a configurable platform service. As in the case of the run-time monitoring of temporal logic properties, the information transfer is optimized to reduce overhead.

The starting and stopping of monitors is the responsibility of an *execution context* (EC), which works as follows. It stores references to the compiled LSC observers, thus it can instantiate a monitor at any time. In general, a monitor is instantiated when there is no monitor being in the initial state. Since there may be several concurrent monitor instances, the EC is used to receive the messages from the running automata and forward these to the appropriate monitors. The EC is connected to the running monitors and receives information about their status (i.e., whenever a monitor is waiting for message or when a message is evaluated). A running monitor is killed if (i) it reached its end state, or (ii) there is another monitor that is in the same state, or (iii) a violation of conditions is detected. In case of a cold condition violation or pre-chart violation only the monitor is killed, while in case of a hot violation the EC also signals an error.

Specific behaviour of the EC is required in the case of various types and activation modes of the LSCs.

- The type of an LSC can be *existential* (for specifying possible or sample runs of the systems) or *universal* (for describing mandatory behaviour for the system). If an universal LSC fails then there is no way that in the future the system will satisfy the specified behaviour. However, if an existential chart fails then there is a chance that another run of the system will be successful. Hence, the EC checks whether at least one trace is reported by the monitor as failed and in case of an existential LSC it also checks whether all traces failed. This latter kind of evaluation is relevant in the system testing use case. It is important that there is a difference between the runs which end with a cold violation (the scenario is aborted) and the runs which end with a hot violation (the scenario is failed).
- The activation mode of an LSC can be *initial*, *invariant* or *iterative*. This property determines how the EC should start the monitor(s) checking the pre-chart. In case of an LSC with initial mode, the EC should start monitors until one monitor is able to leave checking the pre-chart (the behaviour in the pre-chart is matched). In case of an invariant LSC, the EC starts the monitors until the run-time verification is stopped. The iterative mode for existential LSCs is the same as the invariant mode, while for universal LSCs it means that once the pre-chart is matched, further start of monitors is stopped until the main chart is matched or violated (i.e., only one monitor is active).

Finally, we summarize the changes that were necessary in the observer construction in comparison with the algorithm presented in [2]. These changes are due to (1) the restrictions that are effective in the

given application domain of our TA models and correspondingly for the code generator (lack of support for global variables and committed states), and (2) the configuration of the platform services used for communication, and finally (3) the behaviour of the execution context.

- The original construction algorithm duplicates the channels and synchronizations of the system to make the synchronizations observable for the monitor. Our algorithm skips this step, because the notification of the monitor can be implemented in the communication platform service.
- The LSC semantics defines that when a condition-check or an assignment is not associated with a message, then it has to be checked immediately (referenced often as as-soon-as-possible semantics of so-called non-message simregions). For verification, it is needed to create a few committed states in the observer automaton and two global variables in the system. We managed to skip this modification, as the running monitors do not process the synchronization notifications from the checked microcontrollers until there is unhandled non-message condition-check or assignment.
- The original algorithm adds a transition from the last state of the observer automaton to its initial state for handling the invariant mode of the LSC. We skipped this modification, because the different modes of the LSC are handled by the EC.
- Lastly, the observer automaton created by the original algorithm should handle the behaviour that is implicitly allowed by the LSC. This is due to the LSC semantics that allows any synchronization on channels which are not included in the scenario. In our case this is also managed by the EC as it only delivers a notification about the relevant synchronizations to the running observers.

6 Summary and Future Work

In our work we managed to develop an integrated tool for the synthesis of application source code of distributed microcontroller systems from timed automata models, and for the synthesis of run-time verification modules which are able to check the satisfaction of requirements on the running system. We implemented algorithms for constructing (1) monitors for local control flow checking to detect the anomalies of individual controllers, (2) monitors for system-level TCTL verification, and (3) monitors for observing and checking the system behaviour defined as Live Sequence Charts.

We used our tools with the modular mitmót system and we also ported to mbed microcontrollers for a distributed control system containing multiple mbed cards. The porting was relatively easy thanks to the separated definition of platform services in the code generator. The use of configurable platform services allowed to deploy local monitors either as processes executed on the same microcontroller or as separate checkers. The system-level monitors were deployed on a dedicated device using the easy-to-use Ethernet based communication capabilities of the controllers. Both systems were designed and verified in UPPAAL and then our code synthesis tool generated the source code for the controllers. The program code overhead in the checked controllers were acceptable: instrumentation for local control flow checking introduced 3.8% overhead, TCTL and LSC based checking needed about 0.3% and 0.5% more code, this way the instrumentation for all checking possibilities resulted in slightly less than 5% code overhead. The execution time overhead depends on the application to be checked: implementing functions that perform control and communication on the serial bus of the microcontroller, the run-time overhead for all checking was only 12%. However, in case of fast control functions (only setting digital output signals) the time required to provide information to the checker modules dominated the execution time of the application.

In the future we plan to extend the system-level monitoring capabilities to support the testing and run-time verification of context-aware mobile autonomous systems, which requires a new formalism to specify the required properties.

References

- [1] R. Alur and D. L. Dill. Automata for modeling real-time systems. In Michael Paterson, editor, *Proc. of the 17th Int. Colloquium on Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer, 1990.
- [2] S. Balaguer. Specification of properties using live sequence charts. Technical report, Aalborg University, Center of Embedded Software Systems, 2009.
- [3] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks. Uppaal 4.0. In *Proc. of the 3rd Int. Conf. on the Quantitative Evaluation of Systems*, pages 125–126, Washington, DC, USA, 2006. IEEE Computer Society.
- [4] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19:45–80, 2001 April.
- [5] D. Drusinsky. The Temporal Rover and the ATG Rover. In Klaus Havelund, John Penix, and Willem Visser, editors, *Proc. of the 7th Int. SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330, London, UK, 2000. Springer-Verlag.
- [6] B. Finkbeiner and H. Sipma. Checking finite traces using alternating automata. *Form. Methods Syst. Des.*, 24(2):101–127, March 2004.
- [7] K. Havelund and G. Rosu. Monitoring Java programs with Java PathExplorer. Technical report, 2001.
- [8] K. Havelund and G. Rosu. Testing linear temporal logic formulae on finite execution traces. Technical report, 2001.
- [9] M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge University Press, 2006.
- [10] B. Jeszenszky and G. Horányi. Development of distributed embedded systems using formal models and model-based, configurable code generation [in Hungarian]. Technical report, Scientific Students’ Association, Budapest University of Technology and Economics, 2010.
- [11] Java Emitter Templates (JET) framework. <http://www.eclipse.org/modeling/m2t/?project=jet#jet>. [Online; accessed 17-Aug-2011].
- [12] K. Larsen, S. Li, B. Nielsen, and S. Pusinkas. Verifying real-time systems against scenario-based requirements. *Formal Methods*, 5850(2):676–691, 2009.
- [13] The mitmót system. <http://bri.mit.bme.hu/?l=mitmot>. [Online; accessed 17-Aug-2011].
- [14] G. Pintér and I. Majzik. Impact of statechart implementation techniques on the effectiveness of fault detection mechanisms. In *Proc. 30th EUROMICRO Conference*, pages 136–143. IEEE Computer Society, 2004.
- [15] G. Pintér and I. Majzik. Automatic generation of executable assertions for runtime checking temporal requirements. In *Proc. of the 9th IEEE Int. Symposium on High-Assurance Systems Engineering*, pages 111–120, Washington, DC, USA, 2005. IEEE Computer Society.
- [16] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.
- [17] M. Rodríguez, J.-C. Fabre, and J. Arlat. Wrapping real-time systems from temporal logic specifications. In *Proc. 4th European Dependable Computing Conference*, EDCC-4, pages 253–270. Springer, 2002.
- [18] John Rushby. Runtime certification. In Martin Leucker, editor, *Runtime Verification*, volume 5289 of *Lecture Notes in Computer Science*, pages 21–35. Springer Berlin / Heidelberg, 2008.
- [19] Cs. Tóth, Gy. Simon, T. Dabóczy, B. Scherer, L. Kádár, G. Samu, Z. Benesóczky, and G. Péceli. A modular platform for embedded information technology. In *Proc. of the Design and Diagnostics of Electronic Circuits and Systems Workshop*, 2005.
- [20] The UPPAAL tool. <http://www.uppaal.org>. [Online; accessed 17-Aug-2011].