06AE-88

Application Development for Safety Critical Distributed Embedded Systems with Model Verification

Marc Schanne and Dr. Andreas Judt
Software Engineering
Forschungszentrum Informatik (FZI)
schanne@fzi.de, judt@fzi.de

ABSTRACT

The increasing use of distributed applications in real-time and safety critical embedded systems results in the need for functional and non-functional system verification in the design process. This paper proposes model verification as solution to identify concept ional design failures in advance, and to verify model correctness in an abstract level.
With an extension to classical model checking environments like SPIN, or real-time model verification tools like UPPAAL, it is possible to analyze communication in distributed systems and verify design decisions compared to real hardware and system environments like network bandwidth or computing capabilities. With identification of communication points in distributed embedded systems and annotation of non-functional system requirements model verification can be adapted to support application development for safety critical systems in automotive or avionics.

1 INTRODUCTION

The use of distributed, object-oriented applications in the domain of safety critical systems requires a profound verification of used technologies and implementations. In recent years object-oriented technologies get more and more accepted in the field of safety critical and business critical systems [10]. Even if the discussion is controversial [19], the partner in the ongoing research project for High Integrity Java Applications[1] (HIJA) [1, 18] support the development of safety critical and business critical systems with object-oriented techniques and several verification tools.

The main focus in the project is the definition of Architecturally Neutral, high-integrity Real-Time Systems (ANRTS) with use of Java technology for the implementation of networked real-time embedded systems. The addressed application area requires the use of static verification to guarantee the implementation without any failures. To get a bird's eye view of the proposed tool chain section 3 introduces these with respect to the model checking tool presented later in this paper.

- The analysis of concurrent real-time threads for scheduling is provided for periodic and sporadic tasks with rate monotonic techniques;
- Worst-case execution time analysis provides upper bounds for method execution without interaction and blocking for single threads;
- Resource demands related to the different Java memory areas are analyzed for all application's tasks;
- Data-related assertions, pre-, post-conditions, and invariants are verified with a theorem prover adapted from the KeY project [3]; and
- Verification problems such as checking for misuse of RTSJ memory areas or potential null-pointer exceptions are checked with static data flow analysis based on the intermediate representation of the application in the Jamaica compiler [2].

The goal of model checking is differing and allows orthogonally the use of the other results and user annotations. With model checking non-functional timing requirements in networked high integrity systems are verified against real hardware and runtime environments. The proposed tool allows the verification of ANRTS implementations to assist the user in developing for the HIJA network model and his application protocol design.

The strength of model-checking is the verification of control flow with identification of unanticipated interleaving of concurrent threads, parallel execution, and communication. The following subsection discusses this for HIJA in an abstract way and provides a basis for the model checking tool design in section 4.

1.1 INTENDED BENEFITS OF MODEL CHECKING

Particularly for safety critical and business critical systems it is not enough to merely show that a system can meet its requirements (i.e., with extensive tests), it is necessary to show, that a system cannot fail to meet its requirements.

Model checking of finite state machines can verify critical system behavior. With careful simplification, model

1 IST-511718, partially funded by the European Commission in the 6th framework program

checking can identify non-pseudo errors in the implementation of networked safety-critical systems [13]. While formal verification concentrates on small programs due to its complexity, model checking provides a good compromise between testing and formal verification methods. Failures in a simplified program do not necessarily occur in the original: they indicate positions in the program where errors may cause a system failure. The challenge of code simplification is to avoid state explosion while errors in the model should assist the user to find errors in the original implementation.

The approach in the HIJA project simplifies a Java implementation of a distributed safety critical embedded system. The section 4.1 introduces the method used to parse and analyze Java source code with additional user's annotations. We concentrate on time constraints for real-time network communication and real-time method execution. With simplification and automated generation a set of timed automata, which describe the system's real-time constraints and communication patterns are identified. With additional descriptions of the used hardware these automata can be extended with assertions about the runtime environment with network bandwidth or computing capabilities. Section 4.2 specifies this mechanism and proposes a model generator for code simplification and generic identification of communication protocols. The adherence and feasibility of the complete system with respect to real hardware constraints can then be validated with model checkers for timed automata [17] like UPPAAL [7].

An error in the models can provide a hint for an error in the system's implementation or indicates potentially improper conditions for the used execution environments. In contrast, an error free checking cannot guarantee error-free execution, but this model checking approach can be used in software development tools to find errors in the development phase that have to be tested with prototype hardware. We believe that this approach reduces the number of iterations in the development process of high-integrity real-time systems. Compared to related work – introduced in the next section – the HIJA approach benefits from the integration in a full tool chain for the development of safety critical and business critical systems.

2 RELATED WORK

Software developers choose from three approaches for model checking. The first approach translates a program into a model checker's input language. The second approach is to develop language specific tools, while the third checks programs with a combination of both.

The Bandera project [8] addresses one of the major obstacles in the path of practical finite-state verification of software. Tools like SPIN use a simple programming language as formal description of finite state machines. Programs have to be simplified and translated into languages like PROMELA. The simplification causes the loss of program semantics, for example, computation with float numbers. Bandera uses a simplified Java language as formal description. The use of Java allows developers to concentrate on simplification rather than translation from an object-oriented to a functional language. Bandera integrates existing programming language processing techniques with techniques to provide automated support for the extraction of finite-state models that are suitable for verification from Java source code. Today, the Bandera system cannot extract a model automatically [14]. It requires guidance from a software analyst to simplify the original Java code.

Java PathFinder 2 (JPF) [5] is a system to verify executable Java byte code programs. In its basic form, it is a Java Virtual Machine that is used as an explicit state software model checker, systematically exploring all potential execution paths of a program to find violations of properties like deadlocks or unhandled exceptions. JPF reports the entire execution path that leads to a defect. JPF is enabled developers to find hard-to-test concurrency defects in multithreaded programs. JPF addresses the state explosion problem of model checking with a state collapsing mechanism that creates states on-the-fly. JPF calculates the states from a simplified and predicate-abstracted Java program. State sets are created with the included Bandera model checker. As SPIN used PROMELA as formalization, JPF used Java byte code. The JPF Virtual Machine is optimized for model checking, not for speed of execution. Predicate abstraction can lead to counter-examples that cannot occur from execution. These pseudo-errors have to be checked with the original program.

While JPF and Bandera try to use the full Java language as a formal description, the HIJA model checking approach reduces the Java implementation of safety critical distributed embedded systems to their points of communication. The result of our simplification is a set of timed communicating finite state machines which are checked with the UPPAAL model checker.

3 TOOL CHAIN

The proposed model checking tool is part of a tool chain developed in the HIJA project. The goal of high integrity applications is addressed with several verification tools described in this section. Based on the facility of real-time threads and networks distributed applications can be implemented. Hard real-time applications for safety critical systems need static feasibility analysis for scheduling and network utilization. The execution time for local, non-blocked of single methods can be bound to an upper limit by use of a Worst-Case Execution Time Analysis (WCETA) tool. With Worst-Case Memory Usage Analysis (WCMUA) the application's demand for stack and heap memory areas is analyzed. The KeY tool performs deductive verification of functional properties of the annotated code, and directly integrated in the byte code to native code compiler a static data flow analysis (DFA) based on the used intermediate program representation is provided.

3.1 SCHEDULING AND NETWORK ANALYSIS

The aim of scheduling analysis is the calculation of worst-case response time bounds for each schedulable object, it verifies if these periodic and sporadic threads meet their deadlines. The used standard response time analysis mainly includes three steps: assign priorities, calculate the blocking times, and calculate the response times.

For the addressed distributed applications the calculated blocking time results from local processor sharing and network usage and available bandwidth. For both measurement scenarios the analysis tool builds on the Modeling and Analysis Suite for Real-Time Applications (MAST) [4].

3.2 WORST-CASE EXECUTION TIME ANALYSIS

The WCETA concentrates on the execution time in a single thread. The HIJA analysis tool is build on top of the WCETA tool Gromit [11], which is adapted for the use with machine code in the form of ELF that is provided by the byte code to native code compiler.

3.3 WORST-CASE MEMORY USAGE ANALYSIS

WCMUA is used to calculate maximum stack use for all non-recursive method calls and the maximum amount of allocation in all heap areas. The tool verifies that the worst-case memory usage doesn't exceed the specified and available memory space. To verify the correctness of memory management using region-based allocation as provided by the RTSJ the static data flow analysis in section 3.5 is used.

3.4 THEOREM PROVER FOR FUNCTIONAL ANALYSIS

The KeY tool [3] performs deductive verification of an annotated Java code in order to statically prove its correctness according to its specifications. The specification of functional properties – verified by KeY – follow the design by contract paradigm, and assertions with pre-, post-conditions, and invariants are specified with the Java Modeling Language (JML) [15].

3.5 STATIC DATA FLOW ANALYSIS

Data-flow related verification is provided by a static data flow analysis integrated in the byte code to native code compiler of the JamaicaVM toolbox [2]. The DFA verifies the correctness of region-based memory area allocation rules by RTSJ, the absence of deadlocks in the local application, and the absence of Java runtime error conditions such as class-cast and null-pointer exceptions.

The application wide data flow analysis is basically a simple iterative algorithm that records for each variable in the system the set of all values that this variable may hold in the current implementation analyses only reference values. there is currently no representation of primitive types. The analysis iteratively adds elements to these sets as long as assignments are found that add new values to these sets. The algorithm stops when a fixed point is reached, this is, when there are no assignments left that would add values to a variable that are not yet recorded in the value set for that variable. The DFA has the difficulty of finding a compromise between the accuracy of the analysis and the efficiency of the analysis for non-trivial applications. The implementation avoids state explosion by managing complexity with less accurate analysis strategies.

4 MODEL CHECKING TOOL DESIGN

The proposed model checking tool differs from the other verification tools in the introduced tool chain. It is designed orthogonally and allows the use of the other results as input.

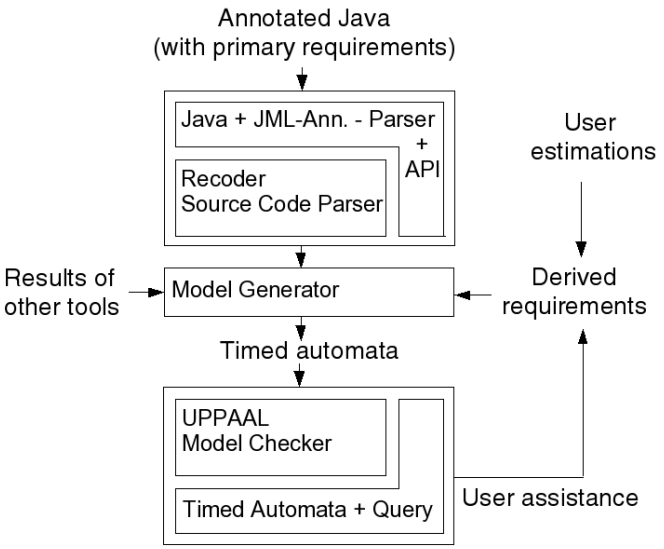

Figure 1: Design of the model checking tool

The figure 1 depicts the component structure of the model checking tool with its relationship to external inputs, that are, Java code with annotated primary requirements and results from other tools or estimations from the application developer.

This section introduces the model checking tool in the HIJA research project for verification of implicit protocols in distributed systems implementations. The design and usage of the tool is introduced by means of the component structure. The three main components are described in the following subsections with respect to their logical tasks and used external tools and APIs.

The tool verifies developer's design decisions by automatic identification of control-flow and interaction protocols out of the application source code. For the second level of use non-functional timing requirements in the ANRTS description are verified against real hardware and runtime environments. . In the first level of use the proposed model checking tool identifies control flow and communication points in the source code by parsing the application. An abstract execution of the control flow extracts a sequence of Java statements and in the

second part this information is aggregated with parsed annotations to simplified timed automata. The last component encapsulates the external real-time model checker UPPAAL and provides generic queries to assist the developer for the implementation of integrity distributed applications.

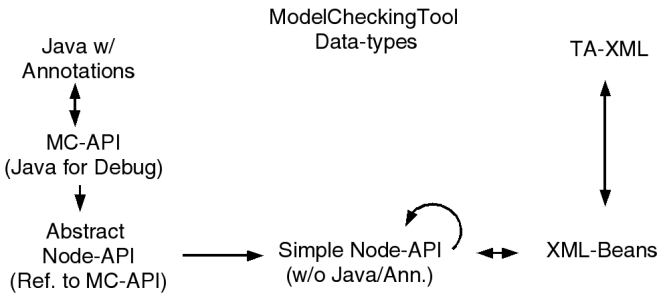

Figure 2: Data-types

A second overview of the tool components gives figure 2 with internal data-types for exchange. The annotated Java code is transformed in several steps into timed automata, encoded with TA-XML, for the model checker UPPAAL. With additional queries these can be checked for common protocol failures and timing problems.

4.1 PARSER FOR ANNOTATED SOURCE CODE

The entry component in the model checking tool represents a Java code and annotation parser (compare figure 1). This parser reads the ANRTS description and generates a sequence of statements, representing an abstract and simplified program execution. Figure 3 shows a fragment of the UML class diagram.

In the first level of use the proposed model checking tool identifies control flow and communication points for the Java RMI protocol as indicated in the class diagram.

With an abstract execution of the control flow the Parser extracts a sequence of Java statements relevant for remote communication.

Loops and conditional statements are returned without really execution and the state in local and instance variables are available over an context object. To avoid state explosion the generator in section 4.2 tries to extract only necessary values.
To support the automated extraction it is possible to annotate each statement with a special comment similar to tags of Java Modeling Language (JML) [15]. The following code fragment demonstrates some possible declarations:

```
04: //@ compoint id = ”init”;
05: s.init(remote_value);
06: //@ maxloopcount = 10;
07: while (true) {
08:   /*@ assert local_val == remote_val;
09:     @ compoint id = ”sync”
10:     @ prev = {“init”, “sync”};
11:     @ maxtime = 20;                @*/
12:   remote_val = s.sync(local_val);
```

Line 05 shows a remote call, and the preceding annotation helps to identify this call and declares an id “init” for further model generation. The next while-loop contains a second communication point with a standard JML functional annotation (i.e., assert) and several protocol information in a non-formal manner. The whole source code, description and transformation of this example to synchronize local states in a client/server application can be found in section 5.

Another annotation in the code snippet shows the second level of use for the model checking tool. The remote call for synchronization is annotated with a required maximum execution time.

```
23:   //@ exec maxtime = $sym1;
24:   val = calcVal();
```

This annotation preceding a normal API method call estimates a maximum execution time for completion. The variable $sym1 is used to reference the used hardware profile with platform dependent values.

This second level of use for the model checking tool is verification of protocol execution under constraints depending on hard coded requirement in the application source code (i.e., line 11). These are called primary requirements and the generated model (compare the next section 4.2) verifies these requirements against platform specific guarantees and parameters.

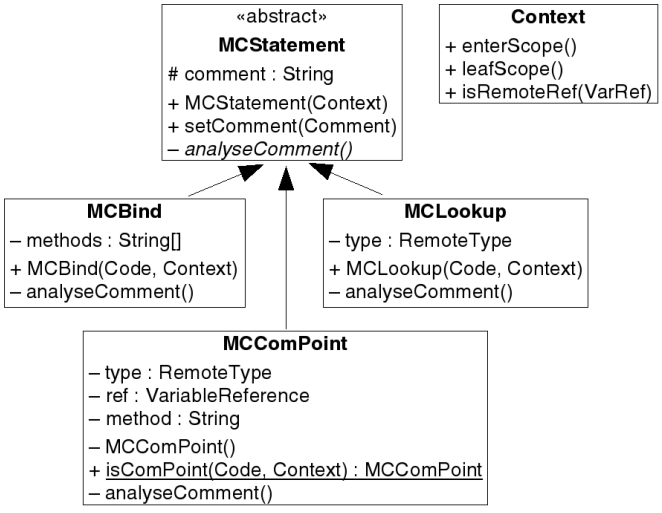

Figure 3: UML class diagram for parsed statements

The platform derived guarantees, parameters, and requirements are part of separate configuration files to allow the implementation of architecturally neutral systems (i.e., ANRTS). The UML class diagram fragment in figure 4 shows an MCAPIMethod class – returned by the ANRTS Parser – with a time value for worst-case execution time, that is annotated similar to example line 23/24.

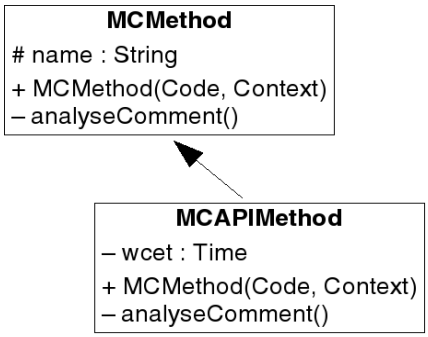

Figure 4: Statement for API method call with WCET

To avoid a state explosion at this point of analysis the application is parsed with main focus on distribution. The program parser identifies potential communication points (remote interaction methods) and simplifies the remaining control flow to statements related to this communication. The state of system variables are available but the following model generation step reduces the used context to communication relevant data, for example, control variables in relevant control flow or parameter and return variables for remote method calls.

To identify protocols in distributed high integrity applications the model checking tool parses the pure Java control flow and remote API use. For traversing and analyze of the source code Recoder [6], a Java framework for source code parsing and meta-programming is used. The supported annotations are similar to tags in the Java Modeling Language (JML) [15] and each statement class (compare figure 3) provides a comment parser to extract additional annotation data.

4.2 MODEL GENERATOR

Because it is difficult to identify the developer's design for a distribution protocol (i.e., correct sequence of method calls or event notifications) automatically, the model checking tool introduces simple annotation semantics to declare this information in an informal manner without huge implementation costs. The approach differs from other languages declaring semantics for concurrent objects [16]. The additional burden for the developer has to be minimized and in HIJA as much as possible results from other verification tools are used.

Each communication point (i.e., remote method call) can be annotated with an identifier and referenced for a simple sequence declaration. The following source code snippet shows two remote method calls from the example in section 5 with these annotations.

```
04: //@ compoint id = ”init”;
05: s.init(remote_value);

09:     @ compoint id = ”sync”
10:     @ prev = {“init”, “sync”};
12:   remote_val = s.sync(local_val);
```

The remote call in line 12 requires a preceding call of the initialization method in line 05 before its first execution.

This logic should also be extractable out of the pure Java source code but for more complex communication sequences in several conditional statements and loops – depending on calculations – that could be hidden for a syntax-driven parser.

This information is also used for the second level of verification usage introduced in the last section. This verification of application execution on real runtime environments itself is divided in two parts. Part 2A simplifies the application environment to optimal conditions, for example, blocking time depending on communication and scheduling of multiple threads, senders and receivers is ignored. In a subsequent analysis 2B with results from other verification tools (i.e., scheduling and networking analysis) this verification can be enhanced for almost realistic scenarios.

To build a suitable model for the system, the source code is transformed in a collection of abstract, state reduced automata. Figure 5 shows a UML class diagram

fragment with elements of this graph, that are, several nodes for each kind of control flow or logic and transitions.

The proposed abstract interpretation of this tool is reduced on remote interactions. This generate finite state automata [12] representing the implemented protocol out of the annotated source code. By extending these automata with timing requirements and guarantees it is possible to verify these timed automata [17] with a suitable model checker, that is, UPPAAL [7].

The main class representing a remote method call with RMI is depicted in the UML class diagram in figure 6.

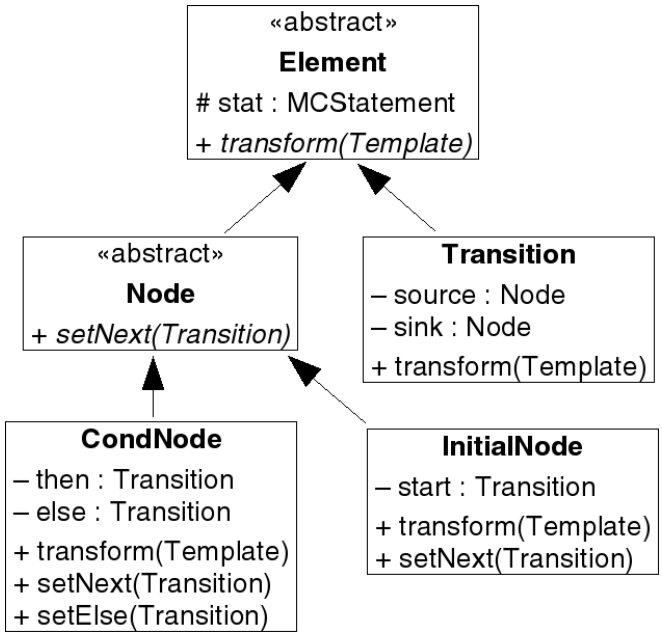

Figure 5: API for abstract node graphs

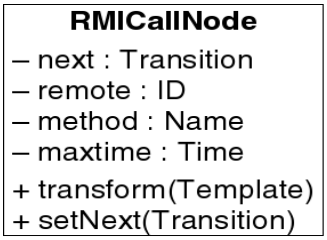

Figure 6: Graph representation for RMI call

The RMI communication point is identified by an object id for the remote object and a method name. The object has to be obtained in a preceding RMI lookup statement and the introduced timing requirements are also saved in the RMICallNode object.

For better understanding figure 7 shows a fragment of the generated model in a simple graph notations with nodes and transitions. The elements can be complex structures represented by a single node as well as superfluous constructs not necessary for the protocol verification. For a RMICallNode in the final timed automaton two nodes and two transactions have to be created, and some of the created nodes could be merged with neighbor-nodes with standard graph and automata algorithms.

Figure 8 shows solely a client/server call, and this synchronous interaction is split in two parts represented with logical channels between two different automata, one for the client and one for the server control flow. The exclamation mark indicates the caller and the question mark the callee side. This concept of channels for finite state – or in this case timed – automata is used to map remote interaction in the collection of timed automata for all available threads.

Relating to figure 2 the model generator uses the sequence of Java statements to create different models of node/transition graphs. An abstract node model with complex or superfluous structures is transformed to simplified graphs, and these are transformed in an object representation for timed automata in XML, that is, TA-XML. After generation this model the TA-XML is serialized to an XML file and used as input for the integrated UPPAAL model checker, introduced in the next section.

4.3 MODEL CHECKER FOR TIMED AUTOMATA

To verify timed automata the proposed model checking tool integrates the UPPAAL model checker [7], and builds an easy to use graphical development environment by integration in the Eclipse IDE with a plugin. In HIJA Eclipse is used as standard development environment and the plugin has access to the ANRTS description and provides an interface for the developer to control the verification. Because the model is extracted in a generic way and is potentially wrong the interface only provides basic queries for verification. With use of the Timed Computation Tree Logic (TCTL) in UPPAAL it is possible to verify the absence of typical protocol failures. With, for example,

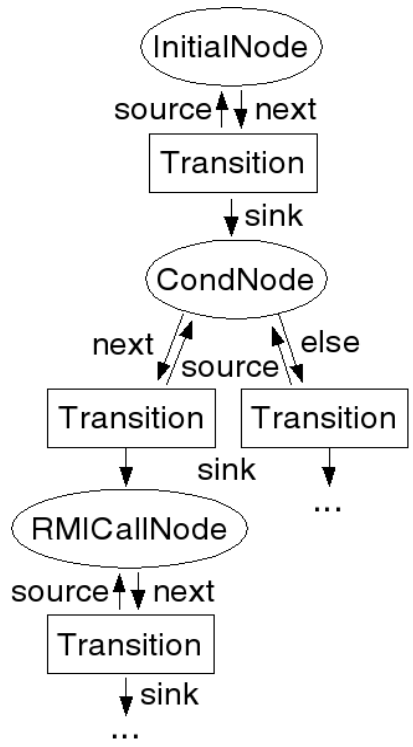

Figure 7: Example nodes and transitions

- `A[] not deadlock;`
- `E<> S.A;`
- `A[] S.A imply (B >= 2 and B <= 4)`

it is verified, that no deadlocks are possible for the whole system, the state A in system S is reachable, and state A implies, that state is reached in the timeframe 2 to 4.

More special, protocol related queries can be verified but that needs more sophisticated interaction with the developer and the plugin tries to limit this to a minimum.

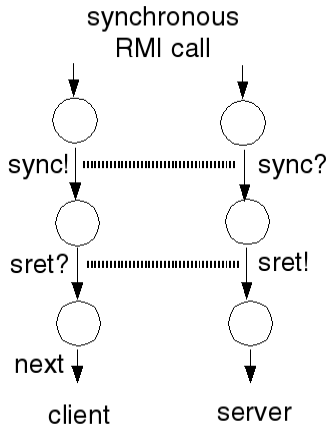

Figure 8: Transformed RMI call

5 EXAMPLE FOR PROTOCOL VERIFICATION

After this rough presentation of the proposed model checking tool in the research project HIJA this section gives a realistic test example to demonstrate the feasibility. In the first implementation step the tool supports RMI method calls, and the example builds a simple client/server application for synchronization of local states in client and server. For demonstration purpose the following Java source code uses simplified RMI and Java syntax.

```
00: // example application to sync
01: // states - client part
02: int local_val = 0, remote_val = 0;
03: Server s = Naming.lookup(“server”);
04: //@ compoint id = ”init”;
05: s.init(remote_value);
06: //@ maxloopcount = 10;
07: while (true) {
08:   /*@ assert local_val == remote_val;
09:     @ compoint id = ”sync”
10:     @ prev = {“init”, “sync”};
11:     @ maxtime = 20;                @*/
12:   remote_val = s.sync(local_val);
13:   local_val++;
14:   if (local_val != remote_val) {
15:     /*@ compoint id = “end”;
16:       @ prev = {“sync”};           @*/
17:     server.destroy();
18:   }
19: }
20: // server part
21: //@ profile = file:url_to_profile
21: int val = -1;
22: public void init(int v) throws RE {
23:   //@ exec maxtime = $sym1;
24:   val = calcVal();
25: }
26: /*@ requires val == cval;
27:   @ exec maxtime = $sym2;          @*/
28: public int sync(int cval) throws RE {
29:   if (val != cval) {
30:     throw new RE(“protocol error!”);
31:   }
32:   incVal();
33:   return val;
34: }
```

Completing the example the following lines show an example hardware profile with platform dependent values on server side.

```
A0: // system description (e.g., capacity,
A1: // performance, throughput)
A1: // list of symbols and values
A2: $sym1 = 10
```

For the distributed example application, client and server implementations are known. The server provides remote methods (i.e., init, sync, and destroy) and the developer has annotated an intended execution sequence (i.e., possible prior ids in line 10). Figures 9 to 11 show fragments of the final TA-XML notation in a graphical representation. The graphs are reduced on basic parts for the distribution protocol.

The two fragments (figure 9 and 10) of server automata shows the main program logic to create and register a remote object and the remote method (i.e., init). Channels are used for local synchronization and remote method calls. The client fragment (figure 11) shows only the communication protocol.

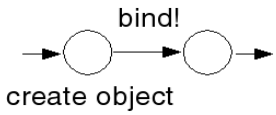

Figure 9: Server main thread

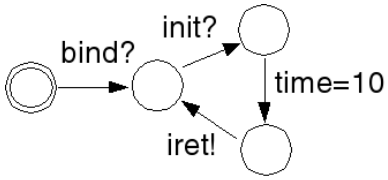

Figure 10: Remote method init

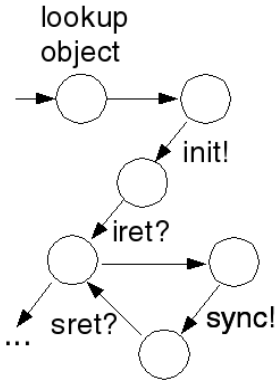

Figure 11: Client protocol fragment

6 CONCLUSIONS

This position paper about the use of model checking in the research project HIJA for verification of safety-critical embedded systems introduces tool design and demonstrates the benefits of model checking for high integrity systems. The verification in two different levels is introduced and the use of architecturally neutral real-time system descriptions with hardware profiles specifying the runtime environment is proposed. We believe that this approach reduces the number of iterations in the development process of high-integrity real-time systems, and by using of existing tools and APIs the development of a user-friendly assistance interface for application development predestinates the future research work in HIJA.

A ACRONYMS

ANRTS: Architecturally Neutral, high-integrity Real-Time Systems
DFA: Data Flow Analysis
HIJA: High Integrity Java Applications
JML: Java Modeling Language
JPF: Java PathFinder
MAST: Modeling and Analysis Suite for Real-Time Applications
RMI: Remote Method Invocation
RTSJ: Real-Time Specification for Java
TA-XML: XML for Timed Automata
TCTL: Timed Computation Tree Logic
UML: Unified Modeling Language
WCET: Worst Case Execution Time
WCETA: WCET Analysis
WCMUA: Worst Case Memory Usage Analysis

REFERENCES

1. HIJA. High Integrity Java Applications. Project Website. http://www.hija.info, 2004.
2. aicas GmbH. The Jamaica Virtual Machine. http://www.aicas.com/products/jamaica.html, 2001-2005.
3. The KeY Project. Project Website. http://www.key-project.org, 2005.
4. MAST. Modeling and Analysis Suite for Real-Time Applications. Project Website. http://mast.unican.es/, September 2005.
5. Java PathFinder. Project Website. http://javapath-finder.sourceforge.net, 2005.
6. Recoder V. 0.75. Project Website. http://recoder. sourceforge.net, 2005.
7. UPPAAL. Tool Website. http://www.uppaal.com, 2005.
8. Bandera. Bandera Tool Set. Project Website. http://www.bandera.projects.cis.ksu.edu, 2005.
9. UPPAAL2k: Small Tutorial, October 2002.
10. Gwinn, J. M. Object-Oriented Programs in Realtime. ACM SIGPLAN Notices 27 (February 1992), 47-52.
11. Hergenhan, A. and Rosenstiel, W. Static Timing Analysis of Embedded Software on Advanced Processor Architectures. In Proceedings of the Design, Automation and Test in Europe (DATE '00) (2000), p. 552.
12. Holzmann, G. J. Design an Verification of Computer Protocols. Prentice Hall, November 1990.
13. Holzmann, G. J. The SPIN Model Checker. Addison-Wesley, 2003.
14. Corbett, J. C. et al. Bandera: Extracting Finite-state Models from Java Source code. In Proceedings of the 22th International Conference on Software Engineering (2000), 439-448.
15. Leavens, G. T. et al. JML Reference Manual, July 2005.
16. Lerner, R. A. Specifying Objects of Concurrent Systems. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1991.
17. Alur, R. and Dill, D. A Theory of Timed Automata. In Theoretial Computer Science. April 1994, pp. 183*236.
18. HIJA partners. D8.1 - HIJA White Paper. Tech. rep., The Open Group, June 2005.
19. Zalewski, J. Object orientation vs. real-time systems. Response to Alan C. Shaw's contribution. International Journal of Time-Critical Computing Systems 18 (2000), 75-77.