Electronic Communications of the EASST
Volume 30 (2010)

International Colloquium on Graph and Model Transformation On the occasion of the 65th birthday of Hartmut Ehrig (GraMoT 2010)

Model Transformations to Mitigate the Semantic Gap in Embedded Systems Verification

Björn Bartels, Sabine Glesner, Thomas Göthel

15 pages

Guest Editors: Claudia Ermel, Hartmut Ehrig, Fernando Orejas, Gabriele Taentzer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Model Transformations to Mitigate the Semantic Gap in Embedded Systems Verification

Björn Bartels, Sabine Glesner, Thomas Göthel

Software Engineering for Embedded Systems Group
www.pes.tu-berlin.de
Berlin Institute of Technology (Technische Universität Berlin)

Abstract: The VATES project addresses the problem of verifying embedded software by employing a novel combination of methods that are well-established on the level of declarative models, in particular process-algebraic specifications, as well as of methods that work especially well on the level of executable code. Beginning with executable code, we (automatically) extract a model in the form of a process-algebraic system description formulated in Communicating Sequential Processes (CSP). For this low-level CSP description, we can prove that it refines a high-level CSP specification which was previously developed. To relate the (Low-Level Virtual Machine) LLVM code with the low-level CSP model we designed an operational semantics of LLVM. In ongoing work we investigate the extraction algorithm with respect to preservation of semantics. Thereby, we are finally able to prove that given LLVM code formally conforms to its high-level CSP-based specification. In this paper we give an overview of results of VATES so far and show that this approach has the potential to seamlessly integrate modeling, implementation, transformation and verification stages of embedded system development.

Keywords: (Timed) CSP, LLVM, Model Extraction, Theorem Proving

1 Introduction

Embedded systems are often employed in safety-critical areas. Their correctness is therefore extremely important in order not to endanger human lives or risk high financial losses. However, the correctness of these systems is difficult to ensure. A particular challenge is that they are highly concurrent and that non-functional properties such as the satisfaction of real-time constraints play an important role. Although there exist well-established techniques to verify abstract specifications of such systems, the verification of their actual implementations, e.g. in C++, is still an open problem.

The VATES[1] project investigates exactly these questions. It starts from the hypothesis that software in embedded systems can be characterized by certain structures (distinguished e.g. by the processes and their pattern of communication) that characterize its mode of operation and that need to be retained when transformed into executable code. We investigate these structures by taking the BOSS [MBK06] operating system as an example. BOSS is a relatively small oper-

[1] VATES=Verification and Transformation of Embedded Systems, funded by the German Research Foundation (DFG). Web: https://group.swt.tu-berlin.de/vates/

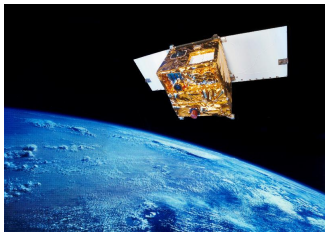

Figure 1: The Satellite BIRD

ating system that has been developed at the Fraunhofer Institut FIRST, Berlin. It is operational since several years within the satellite BIRD[2] (see Figure 1) of the DLR[3] designed for early fire detection. It needs to cope with high performance requirements while only featuring small resources. Such small satellites are very interesting as they have the advantage to be relatively cheap and nevertheless to be very powerful at the same time. Since BOSS has been designed with the goal of verifying it in mind, it is an ideal case study for the VATES project.

We propose a novel approach that makes well-established formal verification techniques for declarative process-algebraic specifications applicable to low-level software programs. To this end, we designed an algorithm that extracts a low-level CSP model from the LLVM compiler intermediate representation [KH09]. On this basis it can be shown that a given low-level model refines a high-level model which is also given in a CSP-based formalism. The high-level specification may be investigated using our formalization of Timed CSP in the Isabelle/HOL theorem prover. It comprises a formalization of its operational semantics, several variants of bisimulations and an investigation of coalgebraic invariants which can be used to state certain liveness properties. Currently, we are using this formalization in the context of so-called parameterized systems to verify infinite-state systems. The semantic gap between the intermediate code and the low-level CSP model is closed by an investigation of the extraction algorithm. Therefore, we defined a formal operational semantics of the Low Level Virtual Machine (LLVM) intermediate language [LA04]. The semantics is especially well-suited for the verification of embedded systems because it includes a memory model and a notion of non-determinism. Furthermore, we establish a bisimulation relation between LLVM and CSP models. With that, we can prove that a given LLVM program is a correct implementation of a given CSP model. We plan to verify that our extraction algorithm preserves the semantics of LLVM using this bisimulation.

The remainder of this paper is structured as follows. In Section 2 we briefly introduce Timed CSP, LLVM and the Isabelle/HOL theorem prover. These constitute the main formalisms and tools that we use in our work. In Section 3 we present the main results of the VATES project so far: a formalization of Timed CSP in Isabelle/HOL, the extraction algorithm giving a low-level CSP model from a given LLVM program and an operational semantics of LLVM in the Isabelle/HOL theorem prover. Related work is discussed in Section 4. Finally, Section 5 concludes this paper.

[2] Bispectral Infra-Red Detection
[3] Deutsches Zentrum für Luft- und Raumfahrt (German Aerospace Center)

$$P := STOP \mid SKIP \mid a \to P \mid a : A \to P_a \mid P;P \mid P \Box P$$
$$\mid P \sqcap P \mid P \underset{A}{\parallel} P \mid P \setminus A \mid P \triangle P \mid P \overset{d}{\triangleright} P \mid P \triangle_d P \mid X$$

Figure 2: Syntax of Timed CSP

2 Used Formalisms and Tools

2.1 TimedCSP

The development chain that we consider in the VATES project, starts with a specification formulated in the real-time process calculus Timed CSP. It is an extension of Hoare's CSP (Communicating Sequential Processes) [Hoa85] with timed process terms as well as timed semantics. Besides the specification and verification of reactive and concurrent systems, this also allows for the verification of timeliness. In the following, we present some of the aspects of (Timed) CSP that are most important for understanding this paper. We refer to [Sch99] for a comprehensive introduction to it.

The syntax of Timed CSP is given in Figure 2. It shares most of the operators with (untimed) CSP: STOP is a process which cannot do anything, SKIP cannot do anything except terminating indicated by the communication of the special event $\surd$, $a \to P$ can first communicate a and then behave like process P. More convenient process operators are e.g. $\Box$, $\parallel$ and $\setminus$ denoting *Choice*, *Parallel Composition* and *Hiding* (of communication channels).

Timed CSP extends the CSP calculus with the timed primitives $P \overset{d}{\triangleright} Q$ (*Timeout*) and $P \triangle_d Q$ (*Timed Interrupt*). Intuitively, the meaning of a *Timeout* is that the process P can be triggered by some (external) event within d time units. If this happens, the *Timeout* is resolved in favor of P. If the time expires without P being triggered, process Q handles this situation, i.e., the *Timeout* is resolved in favor of Q. The *Timed Interrupt* construction has a similar meaning. Here, P can (successfully) *terminate* within d time units, otherwise Q is started.

There exist two main types of semantics which are typically defined in the context of CSP: The denotational (Timed) Failures semantics and the operational semantics which interprets (Timed) CSP as labeled transition system.

For CSP there exist well-established fully-automatic verification tools such as FDR [GRA05] and ProB [LF08]. FDR is well-suited for refinement checking of specifications based on the denotational semantics of CSP. ProB is well-suited to check temporal properties on CSP processes. For Timed CSP there does not exist comprehensive tool support yet. Therefore, we have formalized Timed CSP in the Isabelle/HOL theorem prover as briefly explained in Section 3.1.

2.2 LLVM

The LLVM compiler infrastructure provides a modular framework that can be easily extended by user-defined compilation passes. It also offers a diverse set of predefined analyses and several optimizations that can be used out of the box. It is designed to meet the needs of the development of custom source code transformation and analysis tools. The heart of the compiler infrastructure project is its intermediate representation (IR). It is a typed assembler-like language [LA08],

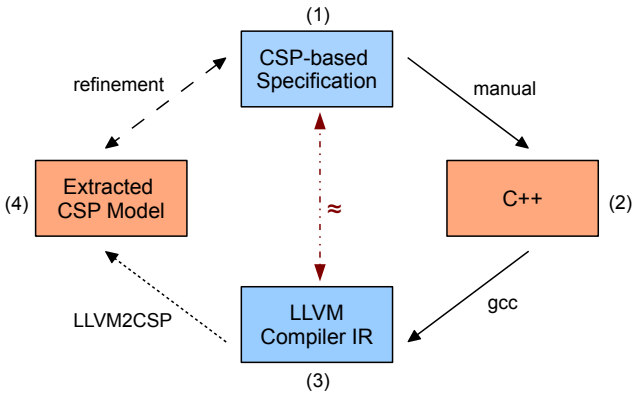

Figure 3: The VATES Proof Framework

which is used internally as the basis for compiler optimizations. The LLVM framework provides gcc-based frontends for a variety of programming languages, including C++. The existence of the gcc-based frontend enables us to adapt our approach, which currently focuses on C++, to a couple of other programming languages with little effort because it is source-language-independent and relies on the LLVM IR only.

2.3 Isabelle/HOL

Isabelle is a generic interactive proof assistant. It enables the formalization of mathematical models and provides tools for proving theorems that are mechanically checked. Isabelle can be instantiated with different so-called object logics. One particular instantiation of it is Isabelle/HOL [NPW02], which is based on Higher Order Logic. The main advantage of HOL is its very high expressive power. Theorem provers based on HOL require a high level of expertise but allow reasoning about models whose state space is too large (or even infinite) to be automatically checked by, say, a model checker. Unlike model checking, proving theorems in a theorem prover like Isabelle/HOL is highly interactive. Specifications have to be designed carefully to enable properties about them to be proved.

3 The VATES Approach

The context of our approach is the VATES project [GHJ07]. Its aim is to develop concepts for verifying the correctness of embedded software. Our goal is to support the verification of crucial properties on all abstraction levels of such a software system, from the abstract specification down to executable code.

The structure of our approach is given in Figure 3. We start with a high-level CSP-based specification (1) where crucial properties are verified. To this end, we have developed a formalization of Timed CSP in the Isabelle/HOL theorem prover and the proof technique of network invariants to verify infinite-state models. This is explained in Section 3.1 in more detail.

Since we deal with embedded applications we want to allow manual code optimizations. We therefore assume that a Software developer implements this high-level specification in a high-level programming language such as C++ (2). To support this task we follow a prototyping approach presented in [KB10]. The high-level language can be further translated into the LLVM intermediate representation (3), e.g. by using the gcc compiler. We chose to relate the abstract CSP model and the intermediate LLVM representation by automatically extracting a low-level CSP model from it (4). This is done with our tool called llvm2csp which is explained in Section 3.2. To show the refinement relation between the high-level and the low-level CSP models, our formalization of Timed CSP or standard tools like FDR2 can be applied.

A crucial part in our approach is to show that the extraction algorithm of llvm2csp preserves the semantics of LLVM. To this end, we developed an operational semantics of LLVM. Due to the simplicity of intermediate languages, this is far more easy than defining a comprehensive formal semantics of e.g. C++. Furthermore, we defined a variant of bisimulation which enables to semantically compare LLVM programs and CSP models. This is presented in Section 3.3.

Altogether we are thereby able to formally relate the high-level CSP-based specification and its corresponding implementation in LLVM. Note that following the overall approach we do not have to formalize the semantics of a high-level programming language like C++ which is known to be a complex task in its own. This becomes even more complicated by the introduction of concurrency. Since we want to verify the whole development chain, the formalization of some intermediate representation is inevitable in each case. So by considering *only* the intermediate language layer circumvents the complex task of formally defining the semantics of C++.

3.1 Verification with Timed CSP and Network Invariants

In a previous paper [GG09], we proposed a formalization of the operational semantics of the process calculus Timed CSP in the Isabelle/HOL theorem prover [NPW02]. This formalization is briefly explained in the following section. Thereafter, we give an overview of current work which is based on the verification of infinite-state systems by using network invariants.

3.1.1 Formalization of Timed CSP

We combined the advantages of specifying real-time systems concisely and of mechanizing correctness proofs for properties of their specifications. We transfered the coalgebraic notions of bisimulation and of invariants to Timed CSP. This allows us on the one hand to relate behaviorally equivalent Timed CSP processes and on the other hand to state invariant behavior of processes. To this end, we formalized the syntax of Timed CSP as inductive datatype and the operational semantics as inductively defined set of triples. It is convenient to define (greatest) bisimulations and (greatest) invariants with respect to state predicates coinductively. We showed that all the considered kinds of bisimulation (strong, weak and weak timed) fulfill the congruence property with respect to the structure of Timed CSP processes. Furthermore, we showed that coalgebraic invariants are well-suited to express certain liveness conditions and that these special invariants are closed under bisimulations. This property is useful for verification as shown in [GG09] in the context of a (rather simple) satellite system specification.

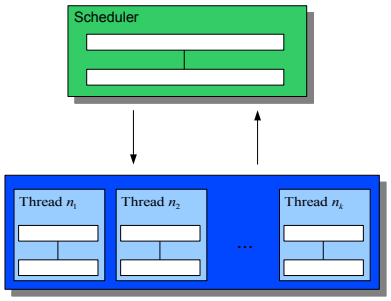

Figure 4: Structure of our Scheduler Model

3.1.2 Parameterized Systems and Network Invariants

In ongoing work we extend this theory by so-called network invariants which are suitable to verify parameterized systems. Our motivation is BOSS's real-time scheduler which we have modeled in Timed CSP. The main structure of the model is given in Figure 4. The overall system includes the scheduler itself and an arbitrarily large number of threads.

The scheduler runs in parallel with arbitrarily many processes representing the threads to be managed. Each thread is characterized by a name and a priority. In order to verify the scheduler system, the specific details of the threads should have no relevance. The scheduler system should, e.g., be deadlock-free in every case, i.e., for every possible list of threads. Thus, an appropriate "induction" on the length of the list should be enough to state the system's correctness. On a certain level of abstraction, all the CSP processes representing threads are homogeneous from the scheduler's perspective. Of only importance is that each thread can communicate with the scheduler (e.g., yield control), and conversely the scheduler with each thread (e.g., give control to a thread). That is why the scheduler system can be seen as a parameterized system. As a consequence, verification of parameterized systems appears to be a promising technique to verify real-time operating systems.

Parameterized systems, as considered here, have the form:

$$N_n = P_0 \odot \underbrace{P \odot \cdots \odot P}_{n}$$

where the variable n (representing a natural number) is the parameter of the system. P_0 is a control process and $P \odot \cdots \odot P$ a network of homogeneous processes. The operator $\odot$ is some kind of parallel composition, which may be equipped with hiding and renaming of communication channels. Network Invariants can be used to verify parameterized systems by dividing the infinite-state verification problem into several (ideally finite-state) verification problems such that automatic verification tools can be used. In this section we focus on the presentation of the results of [WL90] concerning the inductive verification of parameterized systems as this is sufficient here. We have formalized these in the Isabelle/HOL theorem prover. We have, however, also formalized the slightly more general results of [KM95] in Isabelle.

The idea of the "appropriate induction" is formalized in network invariants: To verify whether the system N_n (regardless of the concrete parameter) implements a specification S, the main idea

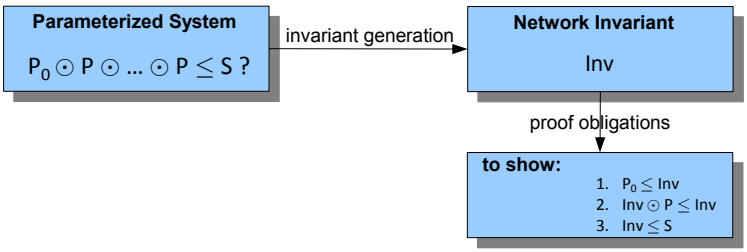

Figure 5: Framework for Verifying Parameterized Systems

is to use an appropriate invariant, which overapproximates each instance such that the abstracted system is still contained in the specification S. In [WL90], it is shown how network invariants can be set up in process algebras like CCS and CSP. There, a general technique for verifying parameterized systems based on network invariants is presented: if one wishes to show that N_n fulfills a certain specification S with respect to a certain implementation relation, i.e., $N_n \leq S$, it must basically be shown (for some process Inv) that $P_0 \leq Inv$, that $Inv \odot P \leq Inv$ and that $Inv \leq S$. The crucial point is to find an appropriate invariant Inv. By induction on n, one can deduce that $N_n \leq Inv$. By proving that the invariant is contained in the specification, i.e., $Inv \leq S$, one can deduce that $N_n \leq S$ by transitivity of the implementation relation $\leq$.

There are two main tasks that have to be performed when working with network invariants for verification. The first is finding a process which may serve as appropriate "network invariant". The second task is showing that the found process *is* indeed a network invariant. In [CGJ97], for example, a technique for finding network invariants based on network grammars is presented. In [GL08], the technique of network invariants is used in the context of timed systems. The approach adopted there consists of two steps. First, a safe abstraction is performed on a given timed system. Thereby, safe means that LTL formulas are preserved under the abstraction. The abstract system is then used as a network invariant, which allows for the verification of the whole parameterized system.

The overall verification flow that we use is given in Figure 5. It is subject to current and future work to integrate invariant generation algorithms and automatic verification tools such as FDR2 [GRA05] and UPPAAL [BY04] into our existing Timed CSP formalization in Isabelle/HOL. In [Oua01] and [DHQ+08] it is presented how Timed CSP can be mapped to tock-CSP[4] or timed automata, respectively. These mappings make the usage of the automatic verification tools FDR2 and UPPAAL available for the verification of Timed CSP processes. In two current master theses we are implementing and evaluating these mappings.

3.2 Relating Low-Level Models and LLVM

In this section we give an overview of the parts of a low-level CSP model which can automatically be derived from a LLVM program using our llvm2csp tool [KH09]. The tool extracts CSP models from concurrent programs. Besides the behavior of the executed threads, the algorithm also uses information about the actual execution platform as explained in the following subsection.

[4] a dialect of (untimed) CSP

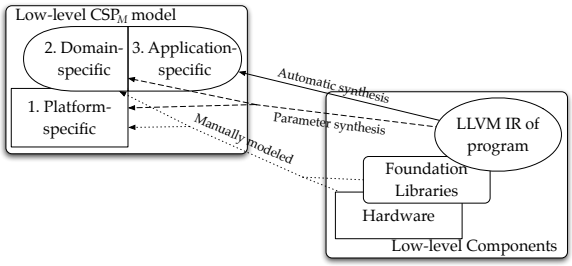

Figure 6: The three Parts of the Low-Level CSP Model

3.2.1 Synthesizing a Low-Level CSP Model

The low-level CSP$_M$[5] model as depicted in Figure 6 not only contains processes, types and channels that are automatically generated from the LLVM IR of a program. It also contains two predefined parts which model the platform- and domain-specific parts of the system under investigation. The platform-specific part comprises the environment model and hardware details, while the domain-specific part encompasses aspects that are common to a domain of applications, e.g. system startup and scheduling. These are provided as foundation libraries that the program builds on. In the long term, platform- and domain-specific models are to be chosen from a library. The two parts are mostly manually modeled but are parameterized so that they can be reused by all applications of the domain they have been designed for. Examples of such parameters are typing information for the channels and the set of thread identifiers. The third part is the application-specific one, which describes the behavior of the threads of a multithreaded program with respect to a set of given variable names, function calls and annotations[6]. We already used the llvm2csp tool successfully to create the low-level model of the scheduler of the BOSS operating system pico-kernel, which we presented in [KBGG09]. Nevertheless we are constantly extending it by further features.

3.2.2 Design of the Low-Level CSP Model

As discussed in the previous section, the low-level CSP model is divided into three distinct parts. The domain- and platform-specific parts are manually modeled but are parameterized. The parameters and the application-specific part are synthesized from the LLVM IR of the program under consideration. Since we aim to use FDR2 for establishing the formal refinement relation between the specification and the low-level model, the Low-Level CSP Model must be designed in a way such that it can be handled by FDR2 as efficiently as possible. To that end, we followed the rules that are given in the FDR2 manual [GRA05], for example, that renaming is to be used in preference to the parameterizing of a process definition.

Fig. 7 shows an example of modeling memory that stores four bit fields. It is constructed of

[5] CSP$_M$ is a machine readable form of CSP extended by concepts from functional programming and is used as input language of the FDR2 refinement checker.

[6] Annotations can be realized using so-called ghost methods and ghost variables. A ghost variable is a variable that is used for verification purposes only and a ghost method is a method that modifies only ghost variables. Ghost code is commonly compiled into the IR for verification purposes but is not part of the final binary.

```
channel read1, write1, read2, write2, read3, write3, read4, write4 : {0,1}
V1 = let V1'(v) = read1!v -> V1'(v) [] write1?x -> V1'(x)
  within read1?x -> V1 [] write1?x -> V1'(x)
V2 = V1[[read1 <- read2, write1 <- write2]]
V3 = V1[[read1 <- read3, write1 <- write3]]
V4 = V1[[read1 <- read4, write1 <- write4]]
WithHeap(P) = (P) [|{|read1, write1, ..|}|] (V1 ||| V2 ||| V3 ||| V4)
```

Figure 7: CSP_M Model of the Heap, taken from [Kle09].

parallel processes ($V1,\ldots,V4$) that model a single variable each. Since the four processes are structurally equal, just one is modeled manually ($V1$). The others are derived from $V1$ by renaming. The parallel composition of the four processes leads to a process that models the memory. It is synchronized with the application specific part P later on using the function (*WithHeap*). The resulting process can be handled much more efficiently by FDR2 than a process that maintains a constant size list of the four variables, for example. We use this concept to model the heap and the stacks of the threads. The process allows to read an arbitrary value from uninitialized memory cells.

Our approach makes strong use of abstraction to reduce the size of the resulting low-level model in terms of reachable states. This includes abstracting the ranges of data types and abstracting away regions of code that do not transitively influence any of the variables from the set of variables to be included in the low-level model. If, for example, concurrent accesses to a shared counter variable have to be proved race-condition-free, it is sufficient to build the model from the accesses to this shared counter and the locks protecting it.

The expressiveness of CSP_M imposes a limiting factor when formalizing the semantics of the LLVM IR because, e.g., floating point arithmetics are not supported. We therefore restrict ourselves to modeling facilities that are available in CSP_M. Our approach currently supports functions, function calls, conditional and unconditional branching as well as integer arithmetic. It builds on a memory model that supports integers, arrays and uninitialized values. Depending on the properties to be proved, we also use the concept of error codes to detect sources of unwanted behavior or to signal situations that were introduced by abstractions during synthesis of the model. An error code is a fresh event $a \notin \Sigma$ and is always used as in the pattern $a \rightarrow STOP$. In [KH09], we use this concept to detect integer overflow that was introduced by abstraction and that did not indicate a real error in the low-level model. A method on the LLVM IR level is translated into a CSP_M function by llvm2csp. The function returns sequential processes, each modeling a single IR operation. These application-specific processes end up in a domain-specific process that models the continuation of the application, possibly including a thread switch. Further details of the application-, domain- and platform-specific models are given in [KH09].

3.3 Verification on the Intermediate Level

In our approach we relate process-algebraic system models to their implementations given in the LLVM intermediate representation. To formally account for the correctness of this relation, we formally defined the operational semantics of the LLVM intermediate language in Isabelle/HOL. We then relate the corresponding labeled transition system (LTS) to the LTS defined by the

operational semantics of the process algebraic model by establishing a weak bisimulation relation between the two transition systems. Using this concept we plan to verify the llvm2csp extraction algorithm in future work.

3.3.1 Operational Semantics of LLVM

We model a semantic state or configuration Conf as a tuple consisting of functions S for the stack, H for the heap and M for the overall memory, as well as an instruction pointer l:

$$\text{Conf} \equiv \langle S, H, M, l \rangle.$$

The crucial part of the semantics is the memory model. Our model is inspired by the model presented in [BL05]. The functions S and H map non-pointer identifiers to their appropriate types and values, while pointer variables are mapped to the types and addresses they refer to. A memory state within a configuration is modeled by the tuple $M = (Addr, B, F, C)$. We define $Addr$ to be the natural numbers $\mathbb{N}$. It is however possible to replace this with types that represent memory addresses more closely. B yields the block size of a given address, while F marks a memory block as free or allocated. Finally, C returns the content stored at a certain address.

In the context of embedded (operating) systems, models are often non-deterministic. The frequent interaction of these systems with the surrounding environment, for example through a sensor that delivers some kind of information, makes the execution highly non-deterministic. Since LLVM natively offers no means to handle non-determinism, we decided to realize bounded non-determinism by annotating the source code.

Sources for non-determinism need to be annotated in the code, for example reading from a memory location where a value delivered from a sensor is stored. The annotation contains the possible range of values that the sensor may return. For the formal semantics, this implies that a transition from a configuration Conf may have more than one successor configuration. For every identifier that is marked as non-deterministic, the annotations define a function $range : Identifier \rightarrow \mathscr{P}(Value)$. The successor configurations differ from the original configuration only in the value of the register %dest_ident, to which the value read from a marked memory location %source_ident is stored. The value is thereby non-deterministically chosen from the set defined by the *range* function.

3.3.2 Bisimulation Relation

A labeled transition system LTS over the alphabet A is defined as a tuple (S, T), where S is a set of states and $T \subseteq S \times A \times S$ is the transition relation. A special label $\tau \in A$ is used as a label for internal transitions.

We associate the set of states S with the possible configurations Conf from the previous section. As labels, we use dedicated names that relate LLVM code behavior to events in the process algebraic specification. We encode information needed for verification purposes on the process algebraic level using this technique. This includes function and system calls, as well as signals to other threads or user input. The names of the labels are annotated to the original source code.

We explain the idea by using the labels of the LTS to encode that and how a certain variable in a code snippet changes. Since we are interested in changes of variables from a set V, the only

```
1       store i32 0, i32* %i, align 4
2       %0 = load i32* %b, align 4                    ; <i32> [#uses=1]
3       %1 = icmp eq i32 %0, 1                        ; <i1> [#uses=1]
4       br i1 %1, label %bb, label %bb1
5
   bb:                                                ; preds = %entry
6       %2 = load i32* %c, align 4                    ; <i32> [#uses=1]
7       %3 = sub i32 %2, 2                            ; <i32> [#uses=1]
8       store i32 %3, i32* %c, align 4
9       %4 = load i32* %c, align 4                    ; <i32> [#uses=1]
10      store i32 %4, i32* %x, align 4
11      br label %bb2

   bb1:                                               ; preds = %entry
12      store i32 5, i32* %x, align 4
13      br label %bb2
   bb2: ...
```

Figure 8: LLVM Code

instruction that produces externally visible behavior in this case is the `store` instruction. For the `store` instruction, we label the edges of the LTS with the value $\%dest\_ident.S(\%source\_ident)$, if %dest_ident is one of the variables to be tracked and $S(\%source\_ident)$ is its new value. For all other instructions we use the τ label indicating silent internal transitions.

The right side of Figure 9 shows the part of the labeled transition system that corresponds to the code snippet from Figure 8. Here, the set V of variables to be tracked is defined as $\{\%i,\%x\}$. First, in line 2 the content of a variable %b, that was marked non-deterministic and that can range over the values 0 and 1 is loaded to the register %0 and compared with the constant 1 in line 3. If the value of %b is equal to 0, the register %1 which holds the boolean value that is evaluated for the following branching condition is set to true. In this case, the execution continues at the block labeled with `bb`, otherwise the execution continues at label `bb1`. At label `bb` in line 6, the content of the variable %c is loaded to register %2, the constant 2 is subtracted in line 7 and the result is stored to variable %c in line 8. Afterwards, the content of the variable %c is loaded to register %4 and stored to variable %x in lines 9 and 10. Execution then continues at label `bb2`. At label `bb1` in line 12, the constant 5 is stored to variable %x and then execution proceeds at label `bb2`.

A set R of tuples is called a weak bisimulation relation if for every tuple $(P,Q) \in R$ holds that for every transition from Q to some Q' labeled with some α there exists a path from P to some P' with the same label such that the tuple (P',Q') is again in R. Here, a path consists of arbitrarily many transitions labeled with τ before and after a transition labeled with α. The converse property must also hold, i.e. every transition from P must have a counterpart from Q. Two states are called weakly bisimilar if there exists some weak bisimulation relation R where the corresponding tuple is in.

The transition system shown in Figure 9 is the disjoint union of the transition systems of CSP and LLVM. Using the previously defined construction rules for the LTS representing the LLVM code, we can show that the LLVM code in Figure 8 is bisimilar to the following process term:

$$P = \%i.0 \rightarrow (\%x.5 \ \rightarrow SKIP \sqcap \%x.7 \ \rightarrow SKIP).$$

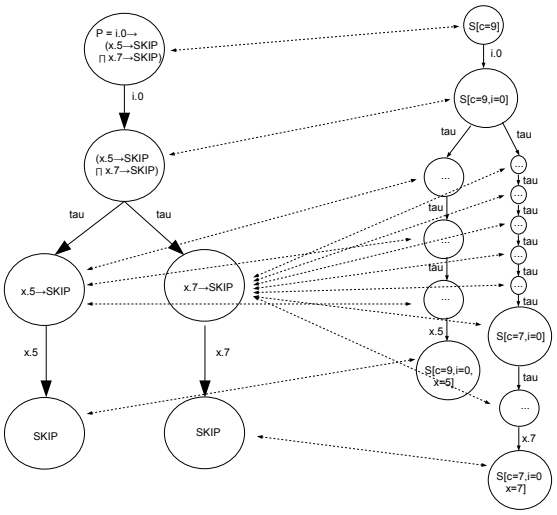

Figure 9: The Bisimulation Relation between the two LTS of CSP (left) and of LLVM (right)

The process specifies that a variable %i is set to 0 and afterwards it is non-deterministically chosen if the variable %x is set to 5 or 7. Figure 9 shows the two labeled transition systems and sketches the bisimulation relation. Note that for every LLVM instruction that does not change any of the variables from the predefined set V, the outgoing edges are labeled with τ. The nodes that are identified by the bisimulation relation are connected by the dashed arrows.

In ongoing work we formalize this notion of bisimulation using Isabelle/HOL to achieve the mechanized verification of the llvm2csp extraction algorithm presented in the previous section.

4 Related Work

The verification of low-level software systems and embedded operating systems has attracted several research projects.

Spec# [BLS04] and VCC [CDH+09] are verification methodologies that translate high-level languages like C# and C to the intermediate language BoogiePL. Necessary verification information is annotated to the source code. To discharge verification conditions, the automatic theorem prover Z3 is used. In contrast to our approach, BoogiePL is not a compiler-intermediate representation but a language tailored for verification purposes. Furthermore, we focus on refinement and transformation of a high-level specification to executable code rather than direct source-code verification.

The AVACS [BDF+07] project deals with the verification of complex (real-time) systems. Verification is carried out at the specification level. Transformations to and the verification of executable code are not considered within that project.

Another project concerned with the verification of a real-time operating system kernel is the VFiasco project [HT05]. Verification is carried out at source code level by defining the denotational semantics for a subset of C++. One of its results is the verification of Duff's Device, but

the techniques were not applied to existing operating system components.
The L4.verified project [EKD+07] uses a refinement approach that proves different layers of abstraction to be consistent. The lowest level of abstraction is given by C code, which is shown to be a refinement of a more abstract design. Neither of these approaches covers concurrency. Furthermore, the transformation to intermediate and executable code is not considered.

Closely related to our work is the verification of a real-time operating system controlling a space satellite using Timed-CSP-Z [SSC03]. Timed-CSP-Z models are translated into a Petri-net-based formalism and the resulting Petri-nets are analyzed. The approach focuses on deadlock detection. Neither refinement proofs nor transformations to code are within the focus of this work.

5 Conclusion and Future Work

In this paper we have summarized the main results of the VATES project so far. We briefly explained the formalization of Timed CSP in the Isabelle/HOL theorem prover and the potential of verification techniques for parameterized systems, namely network invariants, in the context of real-time operating systems. We presented the idea of the extraction algorithm which computes a CSP model for a given LLVM program. Finally, we presented an operational semantics of LLVM and showed how it is possible to formally relate LLVM programs and CSP models with the semantics based on bisimulations.

We are currently working on the integration of timing behavior into the lower levels of our framework. Since timing analyses are already possible on the most abstract layer, we need to extend the llvm2csp tool to generate Timed CSP models. To realize this, it is necessary to augment the LLVM syntax and semantics with information about timing behavior. This includes accounting for processor-specific features like pipelining and cache management.

Since the theory underlying our approach is rather complex we plan to mechanize it entirely using the Isabelle/HOL theorem prover. Thereby we can claim that our transformations are indeed correct.

The results obtained so far are very promising. In future work, we will apply our verification approach to more complex systems, starting with the integration of further system components into our model of the BOSS operating system.

We are convinced that our approach has the potential to enable a methodology that seamlessly integrates the modeling, implementation, transformation and verification stages of embedded real-time system development.

References

[BDF+07] B. Becker, W. Damm, M. Fränzle, E. Olderog, A. Podelski, R. Wilhelm. SFB/TR 14 AVACS – Automatic Verification and Analysis of Complex Systems. *it – Information Technology* 49(2):118–126, 2007.

[BL05] S. Blazy, X. Leroy. Formal Verification of a Memory Model for *C*-Like Imperative Languages. In Lau and Banach (eds.), *ICFEM*. Lecture Notes in Computer Sci-

ence 3785, pp. 280–299. Springer, 2005.

[BLS04] M. Barnett, K. R. M. Leino, W. Schulte. The Spec# Programming System: An Overview. In *CASSIS 2004*. Pp. 49–69. Springer, 2004.

[BY04] J. Bengtsson, W. Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Preti Nets*. Pp. 87–124. Springer, 2004.

[CDH+09] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics. TPHOLs-09, August 2009, Munich, Germany*. Lecture Notes in Computer Science, LNCS 5674, pp. 23–42. Springer, 2009.

[CGJ97] E. M. Clarke, O. Grumberg, S. Jha. Verifying Parameterized Networks. *ACM Trans. Program. Lang. Syst.* 19(5):726–750, 1997.

[DHQ+08] J. S. Dong, P. Hao, S. Qin, J. Sun, W. Yi. Timed Automata Patterns. *IEEE Transactions on Software Engineering* 34:844–859, 2008.

[EKD+07] K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, G. Heiser. Towards a Practical, Verified Kernel. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*. Pp. 1–6. USENIX Association, Berkeley, CA, USA, 2007.

[GG09] T. Göthel, S. Glesner. Machine-Checkable Timed CSP. In *Proc. of The First NASA Formal Methods Symposium*. Pp. 126–135. NASA Conference Publication, 2009.

[GHJ07] S. Glesner, S. Helke, S. Jähnichen. VATES: Verifying the Core of a Flying Sensor. In *Proc. Conquest 2007*. dpunkt Verlag, 2007.

[GL08] O. Grinchtein, M. Leucker. Network Invariants for Real-Time Systems. *Form. Asp. Comput.* 20(6):619–635, 2008.

[GRA05] M. Goldsmith, B. Roscoe, P. Armstrong. Failures-Divergence Refinement - FDR2 User Manual. 2005.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, London, 1985.

[HT05] M. Hohmuth, H. Tews. The VFiasco Approach for a Verified Operating System. In *Proc. 2nd ECOOP Workshop on Programming Languages and Operating Systems*. 2005.

[KB10] M. Kleine, B. Bartels. On Using CSP for the Construction of Concurrent Programs. In *International Conference on Software Engineering Theory and Practice (SETP-10)*. Orlando, Florida, USA, July 2010. in press.

[KBGG09] M. Kleine, B. Bartels, T. Göthel, S. Glesner. Verifying the Implementation of an Operating System Scheduler. In *3rd IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE '09)*. IEEE Computer Society Press, Tianjin, China, July 2009.

[KH09] M. Kleine, S. Helke. Low Level Code Verification Based on CSP Models. In Oliveira and Woodcock (eds.), *Brazilian Symposium on Formal Methods (SBMF 2009)*. Springer, 2009.

[Kle09] M. Kleine. Using CSP for Software Verification. In Mousavi and Sekerinski (eds.), *Proceedings of Formal Methods 2009 Doctoral Symposium*. Pp. 8–13. Eindhoven University of Technology, 2009.

[KM95] R. P. Kurshan, K. L. McMillan. A structural induction theorem for processes. *Inf. Comput.* 117(1):1–11, 1995.

[LA04] C. Lattner, V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California, 2004.

[LA08] C. Lattner, V. Adve. LLVM Language Reference Manual. http://llvm.org/docs/LangRef.html, 2008.

[LF08] M. Leuschel, M. Fontaine. Probing the Depths of CSP-M: A new FDR-compliant Validation Tool. *ICFEM 2008*, 2008.

[MBK06] S. Montenegro, K. Briess, H. Kayal. Dependable Software (BOSS) for the BEESAT Pico Satellite. In *DASIA2006*. Data Systems In Aerospace, Berlin. 2006.

[NPW02] T. Nipkow, L. C. Paulson, M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.

[Oua01] J. Ouaknine. *Discrete analysis of continuous behaviour in real-time concurrent systems*. PhD thesis, Oxford University Computing Laboratory, 2001.

[Sch99] S. Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[SSC03] A. M. Sherif, A. Sampaio, S. Cavalcante. Specification and Validation of the SACI-1 On-Board Computer Using Timed-CSP-Z and Petri Nets. In Aalst and Best (eds.), *ICATPN*. Lecture Notes in Computer Science 2679, pp. 161–180. Springer, 2003.

[WL90] P. Wolper, V. Lovinfosse. Verifying Properties of Large Sets of Processes with Network Invariants. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*. Pp. 68–80. Springer, London, UK, 1990.