Dynamic Composition of Cyber-Physical Systems

Christine Jakobs, Matthias Werner
Operating Systems Group
Chemnitz University of Technology
christine.jakobs@informatik.tu-chemnitz.de
matthias.werner@informatik.tu-chemnitz.de

Peter Tröger
Computer and Information Systems Lab
Beuth University of Applied Sciences Berlin
peter.troeger@beuth-hochschule.de

Abstract

Future cyber-physical systems must fulfill strong demands on timeliness and reliability, so that the safety of their operational environment is never violated. At the same time, such systems are networked computers with the typical demand for reconfigurability and software modification. The combination of both expectations makes established modeling and analysis techniques difficult to apply, since they cannot scale with the number of possible operational constellations resulting from the dynamics. The problem increases when components with different non-functional demands are combined to one cyber-physical system and updated independent from each other.

We propose a new approach for the design and development of composable, dynamic and dependable software architectures, with a focus on the area of networked embedded systems. Our key concept is the specification of software components and their non-functional composition constraints in the formal language TLA+. We discuss how this technique can be embedded in an overall software design workflow, and show the practical applicability with a detailed resource scheduling example.

1. Introduction

Modern cyber-physical systems growingly assist us in our daily life by solving complex tasks, for example in driving assistance, manufacturing or scalable sensing. Faulty execution of those tasks can lead to negative results such as loss of customers, financial loss, or even loss of human lives. Therefore, more and more of such systems are designed with high demands on reliability, availability and safety.

The behavior of software systems is commonly investigated by analyzing predefined static system configurations before the system is going into operation. Ultra-reliable systems, such as airplanes and nuclear power plants, therefore have a fixed setup with strict operational borders and predefined configuration sets. The proven system layout can fulfill all necessary non-functional requirements and provides guarantees about its operational behavior.

Can we map this classic dependability thinking onto the new world of cyber-physical systems?

A crucial aspect in this area is the necessity for dynamics during deployment and execution. This results from the wish for new features, reconfiguration capabilities or bug fixing after the cyber-physical system is deployed. An obvious example is fixing security bugs, which is recurrently needed in all networked software products.

Software update mechanisms today rely on performing functional composition of modules at loading time or run-time. This concept is known as late binding. When these mechanisms are applied in practical cyber-physical systems, the result is an inherently dynamic composition of reliable and proven static software modules. The modeling of such a dynamic reconfigurable system with the traditional dependability analysis methods, such as Markov chains or fault trees, leads to a massive increase of model size and count. The practical result is an unpleasant trade-off decision to be done, between analyzability and reconfigurability of the resulting system. We argue that the underlying issue creating this trade-off is the inappropriate (or missing) formulation of system composition dynamics in formal analysis.

2. Approach

Our proposed approach is to expand the well-known formal reasoning about system behavior to include composition and deployment of software modules more explicitly. The idea relies on the insight that a software architecture is basically a set of rules, or a contract, that some module composition procedure has to follow. This allows to specify the composition rules of the software architecture in a similar strict (formal) way as it is often done with the single code module.

URI: https://hdl.handle.net/10125/60160
ISBN: 978-0-9981331-2-6

To avoid a vast limitation of the system design space, it is reasonable to focus on certain non-functional properties only during the analysis. This can be expressed by according contracts between different system components that are formally analyzed. The fulfillment of these mutual contracts ultimately guarantees a set of non-functional properties for the overall system.

The description of software modules, their contracts and their non-functional demands must be given in a way that allows a proper formal analysis. This is an interesting challenge, since formal methods and especially formal specification languages are normally used to proof behavioral properties of a single algorithm or software component. In our approach, we propose to use the same formal method for specifying all software components and their composition. The resulting higher development efforts for formal specification are later amortized by the ability to dynamically update also software components in mixed-criticality systems, while keeping dependability promises and the predictable behavior of the system. It also explains why proposing some modeling technique alone is not enough – it must be accompanied by a matching requirement specification and design phase, so that the models naturally evolve from the product development.

The chosen formal technique must consider both low and high level aspects of software, meaning that it needs links to both code generation and general architectural patterns. We decided to use TLA+ in our approach, since it allows to specify clear interfaces between system parts, and allows reasoning about non-functional properties of the composed model.

The remaining parts of the article are organized as follows: Section 3 discusses the requirements for a development workflow that supports formalized system composition dynamics. Section 4 then explains why TLA+ is a perfect fit for such an approach. Section 5 takes both the requirements and TLA+ and combines them to a specific development workflow that can be directly applied in practice. After a short discussion of related ideas in Section 6, we show a small example for TLA-based analysis in Section 7.

3. Requirements on the Development Workflow

From the initial discussion about the demands on future cyber-physical systems, we can derive a number of general requirements for their design and development process:

- **R1** Systems must be able to meet high safety demands;
- **R2** Systems should support a high degree of dynamics with respect to the system configuration (i.e., support of software updates, introduction of new features at run-time, or high integration of software);
- **R3** The design process must support the design of, and the reasoning on, non-functional properties;
- **R4** The design process must be efficient and feasible from an engineering perspective.

Let us shortly discuss these requirements.

R1 says that the results of the design process are systems that can be used for safety-critical applications. We therefore restrict ourselves to the safety problem here, and leave out other dependability aspects such as reliability or maintainability. We do this narrowing based on the assumption that a working approach for safety modeling and analysis can be later ported for dealing with other non-functional aspects.

Safety-critical software (as well as hardware) is commonly subject to certification by legal entities. For those systems, the modeling and analysis methods are predefined in standards and regulations. Such documents also demand the utilization of formal methods for analyzing system safety properties. This cannot be done in late design phases; it must either be applied on the design of the whole system (contradicts R4), or it has to be applied for critical components only (again, during the whole design cycle) while establishing isolation from not-so-critical components (contradicts R2 in conjunction with R3).

R2 is an often observed requirement in the development of modern software systems, for example web applications. Here, the support for dynamics in late development phases and at run-time is fulfilled (with respect to the functional properties) by applying concepts of late binding. This requires the specification of interfaces between software components and between software components and the operating system. In terms of non-functional properties (R3) this is difficult since the impact of such a coupling is hard to define in an apparent way. If in addition R1 requires formal reasoning, the relation between different non-functional properties leads to state space explosions with classic approaches, even for small systems.

As non-functional properties (R3) we regard a.o. several dependability aspects, timing, security properties, or resource consumption. Unfortunately, non-functional properties can rarely be handled in the well-known divide and conquer approach. In addition, the effect vectors of non-functional properties are frequently orthogonal to the functional interfaces, and interaction of non-functional properties between different components happens in a nonlinear matter. This hinders the

fulfillment of R4, as well as of R1 in conjunction with R2.

R4 refers to a number of soft factors in relation to the practical application of an approach. Thus, a new approach shouldn't be "too far" away from existing ones, since developers should be able to exploit gained experiences and skills. E.g., it is a well-known fact that applicative programming languages are rejected by many developers due to unaccustomed concepts, even they could be quite useful in the respective application domains. R4 also requires not to introduce to much overhead, since that tends to be "bypassed".

As one can see from this short discussion, the presented requirements R1—R4 are quite contradicting.

From the discussion of R1 we can conclude that for future cyber-physical systems the use of formal methods is a *conditio sine qua non*. To solve the conflict with R4, such a method should be usable (also) in a constructive way, not (only) in an analytical one. This means that the method must allow to generate software (and possibly hardware), rather than only to analyze properties of the system *after* its design. This thinking is known from the model driven development (MDD) field, where system parts are modeled first and the source code is automatically derived out of these models.

We suggest to use the selected formal method not only to specify a certain system, but also to ensure functional and non-functional properties of that very system. This needs a holistic approach, where the formal method is used as a kind of glue within the whole development process. In style of a construction kit, our approach allows for constructing different architectures with particular non-functional properties. In that way the required dynamic (R2) as well as the safety (R1) is supported already at architectural level, since all new components and all configurations of a system must meet the architectural constraints.

A formal method that supports our approach must allow for different levels of abstractions, allow for the description of functional as well as non-functional behavior, and has to support modularization. For practical applicability, it should offer tools for construction and for verification, and must be acceptable with respect to the classic development process in cyber-physical system design (R4).

4. Choosing TLA+

TLA+ is a specification language for describing a systems behavior as a sequence of states, which are defined as an assignment of values to variables through action predicates [1]. Actions describe the transition between states of the system by changing the values of variables. The temporal level of the formalism describes how sequences of states evolve during execution [2].

Since behavior and (non-functional) properties of the system are both formulated in TLA+, a clear representation of the enabling conditions can be created [3]. TLA+ allows to reason about system properties with a model checker (TLC) [4], and supports formal proofs with TLAPS (R3). It is especially suitable for concurrent and reactive systems (R1).

Most engineers in embedded systems are used to the C programming language or object oriented languages like Java. This makes TLA+ unfamiliar for programmers at first. To overcome this issue, the inventors of TLA+ defined PlusCal. It is close to the syntax of the C programming language (R4), but still has a direct link to the TLA+ formalism, so that formal models can be directly derived from code [5]. Since there are a lot of information sources for TLA+ and PlusCal available in public, we give no further insight in the approaches themselves here. For further information, the reader is referred to secondary literature, for example [1].

Composability describes the ability for flexible exchange of single software parts, without modifying the system as a whole and loosing its guaranteed properties. Similar to Kopetz [6] and Richling [7, 8], we see composability *not* as a property of a running system, but of its architecture. The latter is regarded as a set of building rules that drive the composition. Then, generally spoken, composability is the ability of that very building rules to enforce certain statements for properties of systems that are built following these rules.[1]

We aim for using TLA+ (a.o.) to design and verify composable cyber-physical software architectures, meaning such building rules. If needed, these rules should enforce safety properties at system level (R1). Some of that building rules will apply at system design-time, while others can also be applied at run-time. Especially the latter may include *composition constraints* that can be formulated in TLA+ with *indicator variables*.

5. Development Workflow with TLA+

Given the stated requirements and the choice for TLA+, we can now sketch a matching concrete development workflow, as shown in Figure 1. The composition constraints are verified at architecture level and specify how the non-functional properties of the system composition relate to each other. This is done in a constructive way to overcome the state-space explo-

[1] There exists a slightly difference between the definitions: while for [6] the architecture has to ensure that a property is not invalidated by system integration once that property is established at subsystem level, [7, 8] distinguish different kinds of composability, for example, it allows also that a property *may* be reached at system level.

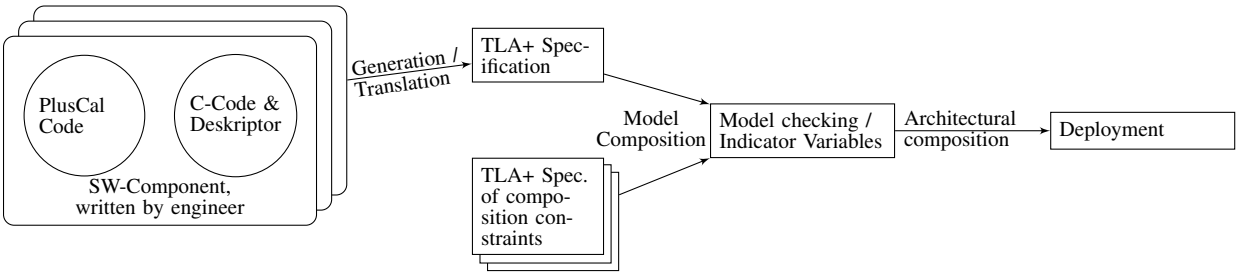

Figure 1. Proposed development approach for dynamic composition of safety-critical real-time systems.

sion problem. Therefore, late-binding in the development process is possible while keeping non-functional properties (R2, R3). Both specification sets (software components and composition constraints) are combined and evaluated using the TLA+ tools.

During the system development, the software components can be first specified as high-level models and then refined during the development process. The non-functional properties of the software components can be defined in the specification. Together with the composition constraints, the resulting software architecture does fulfill the specified non-functional properties, for example timing and dependability.

5.1. Specification of System Components

The root information about the software components can be obtained through different possible ways. Software components can either be specified directly in TLA+ respectively PlusCal, or translated from C code [9] and corresponding descriptors. The latter define the parameters of the component, for example communication and resource demands, that are needed as model parameters. This expectation is not unrealistic, since such data is also needed in all of the established static modeling approaches.

When C-code is translated to TLA+, at first the C code is normalized using a C intermediate language front-end (CIL) before obtaining an abstract syntax tree (AST). At last C2TLA+ generates the TLA+ specification and parameter modules according to the translation rules. The generated specifications can directly be checked by the TLC model checker. This is especially good for backwards compatibility to already existing software components (R4). The necessary input parameters for the model checker can be obtained from descriptors of already existing software components.

5.2. Composition Constraints

Composition constraints are established at architecture level and can be verified at an early stage.

Typically, composition constraints ensure cross-cutting properties (such as timing, security, ...) by guaranteeing invariants with respect to the generic (i.e. application independent) functionalities of the system, such as CPU-scheduling, communication, or resource consumption. Where standards, quasi standards, or best practices exist (e.g., protocol descriptions, scheduling algorithms, ...) composition constraints can be derived respectively from them.

One specific example is CPU resource scheduling. Depending on the operating system capabilities, different scheduling algorithms may be used in different configurations. A formal description of the software modules and their resource utilization, which is commonly available in real-time systems, allows the reasoning about interdependencies and their relation to the final schedule.

In case of communication constraints it may be needed to not only take network protocol descriptions, but also information flow diagrams into account. The results again can be used for the system configuration, for example for routers. Information flow diagrams and information about the safety level of the components can then be used to reason about security properties. By that it is possible to guarantee that critical information is not forwarded to non-secure components (R3).

Since all constraints are described by the same formal method, conflicts can be recognized already at architecture level, especially early in the design cycle (R4).

5.3. System Composition

The system is formally composed from the specified software components, while considering the composition constraints coming from the non-functional system properties. Since we focus on real-time systems, the architecture has to guarantee at least real-time constraints such as timeliness and communication deadlines.

The generated composition result may not be optimal in its resource consumption, but it is possible to it-

eratively define additional constraints. One example are safety properties. Redundancy mechanisms and other fault tolerance strategies generate additional constraints on the deployment in terms of component placement on nodes in the distributed system. One simple example is a triple modular redundancy (TMR) setup, which expects that the triplicated modules are placed on different hardware platforms. This requirement can be formulated as a restriction on the system composition of execution hosts and the redundant modules.

Based on the composition results, it is now possible to configure the operating systems and hardware platforms in the real distributed system. The model parameters of the TLA+/PlusCal code can be transferred into configurations for the run-time environment. The resulting software architecture then does fulfill all defined requirements (R1 – R4).

For software changes during run-time, a classic model checking approach is not enough. However, runtime evaluation of software updates or new features can be done using indicator variables. These variables are responsible for the decision whether a planned system change can be easily implemented, or if the current software deployment has to be changed beforehand. Therefore, a full system analysis through the model checker needs only to be done in two cases.

In the development phase, the model checker is used to state feasibility of the overall system. In case the current deployment is not feasible for updates, the overall system has to be checked to find a new deployment plan.

6. Related Work

TLA+ as specification language is used in a variety of different operational scenarios. Companies like Amazon [4], Intel [10] and Microsoft [11] use TLA+ in their projects to identify subtle bugs and, by that, proof how effective TLA+ can be for big systems. Nevertheless they use the classic approach for specifying only single parts of the overall system. We intent to gather the specifications of multiple software components in the system, together with composition constraints to dynamically compose the system (R2). From the given use case descriptions, it must be concluded that the classic industrial application domain for TLA+ is not safety (R1).

Zhang et al. [3] use TLA+ to specify the functional properties of a fire-fighting system developed for a PLC. They identify flaws in the informal system description taken as a basis for the TLA+ specification. They show why formal specifications lead to more safety in system execution. Although a fire-fighting system is highly safety-critical, it is not subject to frequent run-time updates. Therefore dynamics was not investigated in this case study (R2).

In [2] Grov et al. take a high level TLA+ specification for reasoning about a scheduling strategy in order to implement a more efficient one. They use the transitive character of TLA+ properties to show that the more efficient algorithm preserves the properties of the simpler one. This approach does not reason about safety demands (R1) and dynamic system configuration (R2). Instead they specify a single system part only.

TLA+ is used to reason about real-time systems with different approaches. On the one hand, Faria [12] used TLA+ to reason about real-time system properties of scheduling algorithms for resource access without modeling the explicit timing requirements. Kurki-Suonio et al. [13] as well as Lamport [14] model time explicitly in TLA+ in terms of counters to reason about timing guarantees of the system. These approaches discuss the reasoning about timing as non-functional property (R3) but ignore system composition (R2) and safety (R1).

Following the definition of Kopetz [15], composition is always done with respect to some property. While we attempt to use composition with respect to non-functional properties like safety and timing on the basis of software architectures, most of the times, composition is done in terms of model composition. By that it is meant, that different parts of a system are modeled and then composed through meta-models in white or black box approaches [16] or on simulation layer [17]. For reasoning about system composition and system properties, Richling [8] evaluates the use of Petri Nets for analyzing the composition of embedded real-time systems. His approach does only analyze that the composed system does preserve the required non-functional properties (R3) and safety (R1). We aim to go one step further and compose the system out of the different model parts (R2). Therefore, we combine the traditional approach of model-based development (and by that model composition) with the idea of Richling.

7. Example: CPU Scheduling

A classic case where functional and non-functional properties overlap is scheduling. In real-time systems, timing constraints are a big part of the scheduling approach. To demonstrate the usage of TLA+ for generating (rather than only verifying) system parameters, we choose to demonstrate the generation of a cyclic CPU schedule, as first suggested by [18] for ADA. Please note that TLA+ was already used to verify timing behavior of a given scheduling policy (e.g. [19]), but to our best knowledge we present here the first approach to *generate* a schedule with TLA+.

7.1. Cyclic Executive Scheduling

Beside EDF and RMS [20] as on-line scheduling algorithms, time-driven scheduling is one of the off-line scheduling approaches for real-time systems. Cyclic executive scheduling (also called time-line scheduling, c.f. [21]) is one possible approach where the order of the jobs, as periodic instances of the systems tasks, is computed off-line and stored in a table. One advantage of this approach is that schedulers can be implemented with a very small overhead.

The cyclic executive model assumes periodic tasks, especially tasks that require to be executed on a regular base, once within a given period of time. Each invocation of such a task is called a job. Other than [20], we allow as timing constraints deadlines that may be shorter than the period (but not longer). In fact, we assume deadlines to be not longer than the respective periods for the sake of convenience; it is actually quite simple to extend the approach to allow for general deadlines. For the same reason, we don't include additional resource constraints into our model.

Even if the schedule will run for an arbitrary long time, the scheduling table has a limited size. Since after a certain time interval – called hyperperiod – the scheduling plan repeats, it is sufficient to schedule all jobs only for one hyperperiod. These jobs are then executed completely in one hyperperiod H, called major cycle. [2] The hyperperiod H is divided into frames of size f as minor cycles. In the schedule table, each row contains the jobs of a frame. These jobs can be concatenated as simple function calls, allowing for a reduction of the operating system's overhead. Doing so, timing is enforced only at frame boundaries where the next table entries belonging to the next frame are loaded into the operating system.

While the hyperperiod H is determined as the least common multiple of all task periods in the system – $H = lcm(P_i), i = 1, 2, ...n$ – the determination of the frame size f has three constraints. First of all, the frame size has to be bigger or equal the maximal execution time of the tasks (Equation 1). This is because even the longest task should finish executing within a single frame. To keep the table size small, f must evenly divide H (see Equation 2) and bigger frame sizes lead to less overhead by the scheduler activations. At last, there should be a complete frame between release of a task and the deadline of the task to be sure to detect missed deadlines by the time the deadline arrives (Equation 3).

$$f \geq \max_i(e_i), i = 1, 2, ..n \quad (1)$$

$$f \mid H \quad (2)$$

$$D_i \geq 2f - gcd(P_i, f), i = 1, 2, ..n \quad (3)$$

Once a proper length of the major cycle and of the minor cycles are determined, the actual scheduling problem is to assign each job to a fitting minor cycle. There exist different approaches to perform that assignment, for example by mapping the schedule generation to a network-flow problem (c.f. [22]) and solve this flow problem by a well-known graph algorithm, for example the algorithm of Ford and Fulkerson [23]. We use a different approach as shown in the following section.

7.2. Specification

In the following, we demonstrate how to formally specify the scheduling problem for the cyclic executive approach. We show only key parts of the specification which are needed to understand our idea. The administrative parts for obtaining the results (e.g. specification of $preparetable()$ in Figure 4) are omitted. Optimizations of the algorithm like dynamic frame size determination and job slicing are omitted to keep the specification as short as possible. Therefore we chose a task model of N independent, non-preemptive, periodic tasks. For readers, not familiar with TLA+ and PlusCal, in Appendix A a brief explanation of the used concepts is given.

The input parameters of the scheduling specification is a given task set of periodic tasks, where each task i is described by the following tuple:

$$T_i = (\phi_i, P_i, e_i, D_i) \textit{ where} \quad (4)$$

- ϕ_i phase (to allow offsets of tasks' start)
- P_i period
- e_i execution time
- D_i relative deadline

The different procedures described here are called from the main part of the PlusCal algorithm. To prevent a differing behavior $await$ statements at the algorithms steps are used. These statements can be seen like $wait()$ expressions in programming languages and state that a certain step has been taken beforehand.

For cyclic executive scheduling, as described in Section 7.1 at first the hyperperiod H has to be obtained as least common multiple (lcm) over all execution times (see Figure 2).

After determining the hyperperiod, the frame size f can be obtained according to the constraints of Equations (1)—(3), see Figure 3. For determining f the search is started at the upper bound. Therefore, bigger frame sizes are tested first. Bigger frame sizes lead to longer periods between activation of the scheduler and therefore a smaller system overhead by the scheduler.

[2] In case of general deadlines, the length of the major frame may differ from the hyperperiod

```
gcd(x, y) ≜ CHOOSE i ∈ 1 .. x :
                ∧ x \ %i = 0
                ∧ y \ %i = 0
                ∧ ∀ j ∈ 1 .. x :
                    ∧ x \ %j = 0
                    ∧ y \ %j = 0
                    ⇒ i ≥ j

lcm(b, c) ≜ ((b * c) ÷ gcd(b, c))

hyperperiod: if ( H = 0 ) {
  H := Head(S);
  S := Tail(S) ;

  findlcm: while ( Len(S) ≥ 1 ) {
    H := lcm(H, Head(S));
    S := Tail(S);
   } ;
} ;
```

Figure 2. PlusCal: determining the hyperperiod

```
Max(X) ≜ CHOOSE x ∈ 1 .. Len(X) :
                    ∀ y ∈ 1 .. Len(X) :
                    X[x] ≥ X[y]

procedure Framesize( )
  {
    detmaxforf:
      max := E[Max(E)];
    startfsearch: m := H;
    frange:
      if ( (m ≥ max)
           ∧ (m ≤ H)
           ∧ ((H%m) = 0) )
      {
        k := Len(P);
        detf: while ( k > 0 )
              {
                if ( ((2 * m) −
                      gcd(m, P[k])) ≤ P[k] )
                {
                  f := (2 * m) − gcd(m, P[k]);
                  k := k − 1;
                } else {
                  goto notf ;
                } ;
              } ;
      } else {
        notf: m := m − 1;
        goto frange;
      } ;

    foundf: return;
  }
```

Figure 3. PlusCal: determining the framesize

Before starting to schedule the task set, the tasks are expanded to jobs. For that, a sequence of tuples with the form as in Equation (4) is generated (see Figure 4). The periods and deadlines are expanded till the deadline would be after the hyperperiod.

These jobs then have to be assigned to proper frames. As discussed in Section 7.1 the schedule is frequently generated using flow algorithms. We use a descriptive

```
procedure TasktoJob( )
  {
    startttj:
        k := 1;
        m := 1;
    Appendjob:
        while ( k ≤ Len(E) )
        {
          if ( ((m − 1) * (P[k] + Phi[k])) < H )
          {
            jobs := Append(jobs,
                      ⟨((m − 1) * (P[k] + Phi[k])),
                       E[k], (m * (D[k] + Phi[k]))
                      ⟩);
            m := m + 1;
          } else {
            k := k + 1;
            m := 1;
          }
        } ;
        return;
    }
```

Figure 4. PlusCal: expanding tasks to jobs

```
schedule:
  await (jobs ≠ ⟨⟩);
  l := 1;
  m := 1;
  call preparetable();

schedframes:
  while ( l ≤ Len(jobs)
    ∧   m ≤ (H ÷ f) )
  {
    k := jobs[l];
    if ( ((f * (m − 1)) ≥ k[1])
         ∧ ((f * m) ≤ k[3])
         ∧ ((framel[m] + k[2]) ≤ f) )
    {
      if ( (st[m] = ⟨0, 0, 0⟩) )
      {
        st[m] := ⟨k⟩;
        framel[m] := framel[m] + k[2];
      } else {
        st[m] := Append(st[m], k);
        framel[m] := framel[m] + k[2];
      } ;

      sum_frames := sum_frames + k[2];
      l := l + 1;
      m := 1;
    } else {
      m := m + 1;
    } ;

  } ;
```

Figure 5. PlusCal: assigning the jobs

approach by requiring for every job a frame following three constraints:

- the frame start is smaller/equal the jobs' release time,
- the frame ends after/at the jobs' deadline,

- the frame is not overloaded if the job is scheduled into it (frame load $\leq f$).

As one can see in Figure 5, at every step of the schedule generation, beside verifying that the frame load is lower or equal to the framesize, the current scheduled jobs execution time is added to the frame load.

It can be discussed whether the last constraint is really a constraint for the specification in the algorithm or a property to the behavior. We choose to specify the system by modeling the constraint inside the schedule generation which is easier to understand.

7.3. Generating the Schedule

TLC needs as input the type invariants about the model parameters, the specification which should be proven, and the termination criterion. The type invariants are used to state if the current values of the system variables are correct ones. In the given specification, the hyperperiod, as well as the frame size and the maximal execution time should be natural numbers. The execution time should be non-empty and the tuples of the execution times, periods, deadlines, and phase should be of equal length.

$$
\begin{aligned}
&Spec \triangleq \ \wedge Init \wedge \Box[Next]_{vars} \\
&\qquad \wedge \mathrm{WF}_{vars}(Next) \\
&Termination \triangleq \ \Diamond(pc = \text{“Done”}) \\
&TypeInvariant \triangleq \ \wedge H \in Nat \\
&\qquad\qquad\qquad \wedge f \ \in Nat \\
&\qquad\qquad\qquad \wedge max \in Nat \\
&\qquad\qquad\qquad \wedge E \neq \langle\rangle \\
&\qquad\qquad\qquad \wedge Len(E) = Len(P) \\
&\qquad\qquad\qquad \wedge Len(E) = Len(D) \\
&\qquad\qquad\qquad \wedge Len(E) = Len(Phi) \\
&Term \triangleq \ \Diamond\Box(sum\_frames = sum\_jobs)
\end{aligned}
$$

Figure 6. TLA+: specification, typeinvariant and termination

In TLA+ the specification constraint is always composed out of three different parts (see Figure 6). The $init$ statement specifies which initial values are given to the system variables. As a conjunction of the different actions the $next$ statement specifies the possible system steps. To allow stuttering but keeping the system variables in the same state, the $next$ statement is always true or otherwise the variables are kept unchanged. Fairness information can be applied as third part of the specification constraint to reason about liveness properties.

On termination it is necessary to verify that every job is scheduled which means that the sum of all job execution times is equal to the load of all frames.

E = $\langle$11, 10, 5 $\rangle$
D = $\langle$25, 50, 100$\rangle$
P = $\langle$25, 50, 100$\rangle$
Phi = $\langle$0, 0, 0$\rangle$
Jobs:
- $J_{1,1}$ = $\langle$0, 11, 25$\rangle$
- $J_{1,2}$ = $\langle$25, 11, 50$\rangle$
- $J_{1,3}$ = $\langle$50, 11, 75$\rangle$
- $J_{1,4}$ = $\langle$75, 11, 100$\rangle$
- $J_{2,1}$ = $\langle$0, 10, 50$\rangle$
- $J_{2,2}$ = $\langle$50, 10, 100$\rangle$
- $J_{3,1}$ = $\langle$0, 5, 100$\rangle$

Figure 7. Model parameters and job list of the example task-set

7.4. Generating an Example Schedule

For evaluating the approach for specifying the composition constraints for CPU scheduling we use different task sets. They are given as model parameters with different tuples for the execution times, periods, deadlines and phase. At the end of the specification run, the results are printed on the console. Figure 7 shows the model parameters for a small example of only three tasks as well as the expanded job list. For this task set the model checker determines a hyperperiod of 100 and a framesize of 25. The resulting scheduling table is shown as a flowchart in Figure 8.

In case of task sets which are not possible to schedule, the evaluation will fail. This means, that the analysis will print the first job which is not possible to schedule. If this happens, the task has to be reviewed and either jobs have to be sliced or a different deployment scheme has to be chosen.

7.5. Discussion

We use a formal specification language which allows high-level as well as low-level specifications for our proposed approach. By that, we support all types of non-functional demands that can be expressed in formal statements. On the first view, TLA+ has a high entry point for new users but with a few basic definitions, the code itself is readable. Through PlusCal as algorithm language, it becomes typically very easy to start with.

TLA+ allows to write specifications also for complex data types. Two contrary types of data representation can be used - either sets or tuples. Following R4 we decided to use tuples instead of sets for data representation. Tuples allow to access elements like in C for arrays. Also the standard modules for manipulating sequences and functions can be used with tuples. Utilizing set theory the specification in Section 7 may be shorter, but not that easy to understand for novice users.

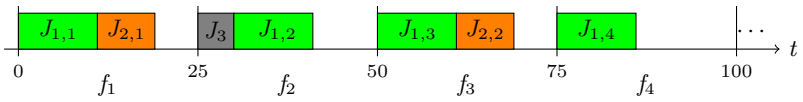

Figure 8. Flowchart of the example task-set from Figure 7

The approach is an unconventional way for using a specification language. We turned TLA+ upside down to generate schedule tables and therefore the needed settings for the operating system scheduler rather than verifying the algorithm itself. This satisfies R2 and R4 as it leads to an approach for dynamic system composition which is similar to commonly used programming languages.

We did not show a complete use case including the dynamics for the reason of space. The presented specification generates a schedule table for a given task set. This task set arises out of the software components which need to be composed in the overall system. For a complete composition use case, showing also dynamics at run-time not only CPU-scheduling but also other composition constraints like resources have to be taken into account. This has to be left for future work and other publications.

The specification was checked by the TLC model checker with different task sets to verify whether the idea of generating correct schedules is working. In our tests, the model checker showed the correct results in affordable run-times. The proposed development approach in its current version should now be used to verify system properties at development time only. This means, that in case the task set changes, the model checker has to run again, to generate a feasible schedule.

We are aware that the TLC model checker is not applicable for run-time evaluation of the system. Instead, for run-time evaluation identified indicator variables may be used as shown by Richling [8] for EDF schedules. The idea is to identify certain rules by which the architecture may break if they are violated. For EDF this is the case, if the utilization gets higher than 1. The next step is therefore to identify these indicator variables for other scheduling algorithms and system parts (e.g. resource constraints). This enables us to generate specifications which can be analyzed also at run-time.

8. Conclusion and Future Work

In this article, we described an approach to guarantee both, dynamic and safety, for embedded real-time systems by utilizing TLA+ as formal method through the whole development process.

In especially, we consider not only to the systems to construct, but also the “meta” levels: composition and deployment. In this way, we want to allow for a forth and back (or more precise: up and down) shifting of dealing with critical concerns without violating the formal correctness. It is our hope, that our approach helps to overcome the complexity pitfall of common formal approaches.

As a first step, we showed the results of writing a formal specification of composition constraints on CPU scheduling.

The presented specification along with the temporal predicates are just a first proof of concept. For application to real world scenarios, we have to further develop the specification to handle job slicing and dynamic frame size determination.

We also plan to expand the specification to consider resource constraints between tasks. In [12] specifications for the priority inheritance protocol are shown which we will take into account.

Instead of just printing out the results - depending on the target operating system - modules can be written to produce the output fitting to the configuration interface of the operating system scheduler.

Additionally, we are going to develop specifications for constraints on resource consumption and communication behavior. While going through the three dimensions of composition constraints, we define the needed input parameters from the software components. The composition algorithms together with the parameter set will enable us to dynamically compose distributed real-time systems in form of a software architecture which keeps the required non-functional parameters.

A. Used TLA+ / PlusCal Notations

In Section 7.2 we used TLA+ as well as PlusCal notations for our specifications. Although especially the syntax of PlusCal is related to the C programming language, there are some notations which are uncommon. Therefore, we give a brief explanation of the used notations only. For a more deep introduction the reader is referred to [1, 5].

PlusCal itself is a algorithm language which can be directly translated to TLA+ code. This means, that an algorithm with macros, procedures and the main algorithm is specified and afterwards translated into TLA+

code. Since TLA+ has the notation of actions which specify the systems behavior, TLA+ and PlusCal use labels to specify the different behavior steps. In PlusCal these labels are needed at any point which may be seen as a different step, different loop runs for example.

In contrast to the C programming language, in PlusCal while-loops are the only loop concept. On the other hand, compared to C, goto is regarded useful [5].

Macros (e.g. *gcd(x,y)* in Figure 2) are like macros in C, and procedures can be used as functions in C. The difference to C is in terms of *return* statements, which do not have return values. Instead for keeping variable changes after the procedures end, these variables have to be global ones since PlusCal uses call-by-value.

TLA+ provides several predefined modules which may be used in specifications. One of those is the sequences module which is also used in our demonstration of the scheduling algorithm. Sequences are comparable to lists, which is why the module can also be applied to tuples. The operators *Head, Tail* and *Len* out of the sequences module are the same a programmer may expect. *Head* provides the first element of a sequence and *Len* returns the total length. *Tail* returns the sequence with the head removed. The *Append* operator expands the sequence at the end and adds the provided element.

Await prevents the step to be executed, if the expression is evaluated to *False*. A concept which may be confusing at first is the *CHOOSE* operator which is known as $Hilbert's\ \epsilon$. It chooses a value for which the expression equals true [1].

References

[1] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[2] G. Grov, G. Michaelson, and A. Ireland, "Formal verification of concurrent scheduling strategies using TLA," in *2007 International Conference on Parallel and Distributed Systems*, vol. 2, pp. 1–6, Dec. 2007.

[3] H. Zhang, S. Merz, and M. Gu, "Specifying and verifying PLC systems with TLA+ : A case study," *Computers & Mathematics with Applications*, vol. 60, pp. 695–705, Aug. 2010.

[4] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How Amazon web services uses formal methods," *Communications of the ACM*, vol. 58, pp. 66–73, Mar. 2015.

[5] L. Lamport, "The PlusCal Algorithm Language," in *Theoretical Aspects of Computing - ICTAC 2009* (M. Leucker and C. Morgan, eds.), vol. 5684, pp. 36–60, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[6] H. Kopetz, *Real-Time Systems*. Kluwer, 1997.

[7] M. Werner, J. Richling, N. Milanovic, and V. Stantchev, "Applying Composability to Dependable Embedded Systems," in *Proceedings of the International Workshop on Dependable Embedded Systems at the 22nd Symposium on Reliable Distributed Systems (SRDS)*, pp. 20–25, 2003.

[8] J. Richling, *Komponierbarkeit eingebetteter Echtzeitsysteme*. Göttingen: Cuvillier, E, Feb. 2006.

[9] A. Methni, M. Lemerre, B. Ben Hedia, S. Haddad, and K. Barkaoui, "Specifying and verifying concurrent c programs with TLA+," in *Formal Techniques for Safety-Critical Systems* (C. Artho and P. C. Ölveczky, eds.), vol. 476, pp. 206–222, Cham: Springer International Publishing, 2015.

[10] B. Batson and L. Lamport, "High-level specifications: Lessons from industry," in *Formal Methods for Components and Objects* (F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, eds.), (Berlin, Heidelberg), pp. 242–261, Springer Berlin Heidelberg, 2003.

[11] F. Lardinois, "With cosmos db, microsoft wants to build one database to rule them all." https://techcrunch.com/2017/05/10/with-cosmos-db-microsoft-wants-to-build-one-database-to-rule-them-all/.

[12] J. M. S. Faria, *Formal Development of Solutions for Real-Time Operating Systems with TLA+/TLC*. PhD thesis, Porto, 2008.

[13] R. Kurki-Suonio and M. Katara, "Real time in a TLA-based theory of reactive systems," in *1998 First International Symposium on Object-Oriented Real-time Distributed Computing, 1998. (ISORC 98) Proceedings*, pp. 186–195, Apr. 1998.

[14] L. Lamport, "Real time is really simple," *Technical report, MSR-TR-2005-30, Microsoft Research*, 2005.

[15] H. Kopetz, "The Time-Triggered Architecture," in *Real-Time Systems*, pp. 325–339, Boston, MA: Springer US, 2011.

[16] B. Tekinerdogan, "A Meta-Model for composition techniques in object-oriented software development," in *Composability Issues in Object-oriented Software Development Workshop, ECOOP'96*, 1996.

[17] S. Kasputis and H. C. Ng, "Model composability: formulating a research thrust: composable simulations," in *Proceedings of the 32nd conference on Winter simulation*, pp. 1577–1584, Society for Computer Simulation International, 2000.

[18] T. P. Baker and A. Shaw, "The cyclic executive model and ada," in *Proceedings. Real-Time Systems Symposium*, pp. 120–129, Dec 1988.

[19] Z. Liu and M. Joseph, "Verification, refinement and scheduling of real-time programs," *Theoretical Computer Science*, p. 34, 2001.

[20] C. L. LIU and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard- Real-Time Environment," *Journal of the ACM*, vol. 20, pp. 46–61, Jan. 1973.

[21] G. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Real-Time Systems Series, Springer Science+Business Media, 2011.

[22] J. Blazewicz, "Selected topics in scheduling theory," *Annals of Discrete Mathematics*, vol. 31, pp. 1–60, 1987.

[23] L. R. Ford and D. R. Fulkerson, *Maximal Flow Through a Network*, pp. 243–248. Boston, MA: Birkhäuser Boston, 1987.