

Verification Architectures for Real-time Systems^{*}

Johannes Faber

Department of Computing Science, University of Oldenburg, Germany
j.faber@uni-oldenburg.de

1 Introduction

In the analysis of realistic systems, one has to cope with different and heterogeneous dimensions that have to be modelled and ideally automatically verified. Real-world systems, e.g., the European Train Control System (ETCS) [1], are determined by process and communication aspects, by rich data structures, and by real-time behaviour. In [2] the ETCS system is modelled using the combined specification language CSP-OZ-DC (COD), which is designed to deal with these system dimensions; a verification approach for COD against Duration Calculus (DC) [3] formulae is provided. But unfortunately, realistic systems are most often too complex to be verifiable fully automatically. So, further decomposition methods are necessary. [2] provides an intuitive manual decomposition that splits the system and a global safety property according to an abstract behavioural protocol. It divides the system runs into several phases (e.g., braking phase, running phase, etc.) with local properties (defined as DC formulae) that hold during these phases. Once the desired property's correctness for such a protocol is established, one only has to verify that the local properties are fulfilled by the system model to guarantee correctness of the global property.

The aim of this conceptual work is to generalise this approach. We extend the specification language CSP [4] by data constraints and undefined processes and show that it is suited to specify those protocols. We introduce a sequent-style calculus over this CSP extension that allows for establishing desired properties under local real-time assumptions. All concrete specifications that are instantiations of abstract protocols and for that the local assumptions are valid automatically inherit the desired properties. With a simple proof rule (that we do not present here) it is possible to show efficiently that a concrete specification is such an instantiation. The correctness of the local assumptions can be shown using established methods for the assumptions' logic. This integration of an operational language to describe protocols and a declarative (real-time) language to describe local properties of a system to simplify verification of large systems distinguishes our approach from standard refinement/implementation approaches, e.g., CSP refinement [5] or data refinement for Z [6]. Hence, we call this combination of abstract protocol and local assumptions *Verification Architecture*.

We summarise our contributions: (1) We provide a new conceptual approach on how to use design patterns, called Verification Architectures (VA),

^{*} This work was partly supported by the German Research Council (DFG) under grant SFB/TR 14 AVACS. See <http://www.avacs.org> for more information.

as a decomposition technique to enable verification of large systems. (2) We introduce a CSP dialect with data, undefined process parts, and local real-time assumptions for the specification of VAs. (3) A new sequent-style calculus over this CSP dialect allows for the verification of desired properties of those VAs. (4) Using a train control system motivated by the ETCS similar to the example from [2], we provide evidence that our method enables the automatic verification of a system that is too large to be verified without decomposition techniques.

The paper is structured as follows. Section 2 explains our approach formally. We introduce the CSP extension and, exemplarily, some sequent calculus proof rules in Sect. 3. Section 4 concludes with experimental results and related work.

2 General Approach

Let $va(\bar{p})$ be an abstract behavioural protocol depending on a vector of parameters \bar{p} . We will use CSP processes with data for the specification of those protocols. Additionally, we consider assumptions $asm_1(\bar{p}), \dots, asm_n(\bar{p})$ over va that also depend on the parameters – here the assumptions are dense real-time properties that are given as DC formulae. Our aim is to show that a global safety property $safe(\bar{p})$ is valid for every possible model that is an instantiation of the abstract protocol va with assumptions $asm_1(\bar{p}), \dots, asm_n(\bar{p})$.

To apply our approach, we have to show in a first step that the architecture together with the assumptions is correct for all possible parameter valuations:

$$(\forall \bar{p} \bullet va(\bar{p}) \wedge \bigwedge_{i=1, \dots, n} asm_i(\bar{p})) \models safe(\bar{p}) \quad (1)$$

This verification task to verify the correctness of the abstract parametric model va is for realistic systems not necessarily easy (in general, it cannot be done by model checking) and we will provide proof rules for the verification. But once it is verified, this result is reusable as all instantiations of this architecture inherit the correctness property automatically. We only have to show that it is an instantiation of the abstract CSP protocol and that the local assumptions asm_1, \dots, asm_n are valid, which is due to their locality easier than to verify the global property $safe$ directly. To be more concrete, we consider an instantiation $cod_C(\bar{p}_0)$ of the abstract protocol va , where \bar{p}_0 represents an instantiation of the parameters. As specification formalism, we use the parametric, combined specification language *CSP-OZ-DC* (COD) [7,8,2], since it is in line with the focused system class of complex, heterogeneous real-time systems. We now apply the result of the architecture’s correctness from (1) to conclude the correctness of the concrete model cod_C . Firstly, we have to show that every trace of cod_C from the trace set $[[cod_C]]$, is also a trace of va , i.e., $[[cod_C]] \subseteq [va]$. This relation can be shown syntactically for a specific class of instantiations. Thus, it is easy to verify. Secondly, we have to show that the assumptions are valid for the concrete specification:

$$cod_C(\bar{p}_0) \Rightarrow \bigwedge_{i=1, \dots, n} asm_i(\bar{p}_0) \quad (2)$$

This can be done by applying existing model checking techniques [2] for COD and DC. With this, our approach yields that the desired safety property is valid for the concrete model. We argue that this proposition is correct. From (1) we can conclude (3), due to $\llbracket cod_C \rrbracket \subseteq \llbracket va \rrbracket$ it then follows (4), and with (2) we get the desired property $cod_C(\overline{p_0}) \models safe(\overline{p_0})$.

$$va(\overline{p_0}) \wedge \bigwedge_{i=1, \dots, n} asm_i(\overline{p_0}) \models safe(\overline{p_0}). \quad (3)$$

$$cod_C(\overline{p_0}) \wedge \bigwedge_{i=1, \dots, n} asm_i(\overline{p_0}) \models safe(\overline{p_0}) \quad (4)$$

We summarise that if a correct Verification Architecture is given, we only have to show that, firstly, a model is actually a concrete instantiation of the VAs abstract protocol and, secondly, the model fulfils the architecture's assumptions. Then we can conclude the correctness of the entire model.

3 Sequent Calculus for CSP Processes with Data

In this section, we give a short idea of our CSP extension and its embedding into the dynamic logic dCSP that allows for specifying and verifying VAs.

To be able to specify VAs, we need a high degree of freedom to handle general patterns of parametric systems with data. To this end, we introduce an extension to CSP with data constraints to define state changes and a new construct, so-called *undefined processes*. Undefined processes are special processes that allow the occurrence of arbitrary events except for events from a fixed alphabet and arbitrary changes of variables except for variables from a fixed set. Undefined processes can terminate and may be restricted by constraints from an *arbitrary* logic (at least, if this logic has the same semantical domain as CSP with data). On the level of CSP, these constraints are handled as black boxes that restrict the possible behaviour of a process.

Definition 1. *The syntax of CSP processes with data and undefined processes over a set of events $Events$, variables Var , and formulae $Form_\Sigma$ is given by*

$$P ::= \text{Stop} \mid \text{Skip} \mid (a \bullet \varphi) \rightarrow P \mid P_1 \square P_2 \mid P_1 \parallel P_2 \mid P_1 \parallel_A P_2 \mid P_1 \circlearrowright P_2 \mid X \\ \mid (\text{Proc}_{\setminus A, V} \bullet_{ext} F) \mid (\text{Proc}_{\setminus A, V}^\infty \bullet_{ext} F)$$

where $a \in Events$, $A \subseteq Events$, $V \subseteq Var$, $\varphi \in Form_\Sigma$, and F is a constraint in an external logic *ext*.

In this definition, a difference to the standard CSP definition is that we have constrained occurrences of events $a \bullet \varphi$. As formulae we consider many-sorted first order formulae with predicates and function symbols from a signature $\Sigma = (Sort, Func, Var, Par)$ with primed and unprimed variables Var , parameters Par , and functions $Func$ with sorts from $Sort$. The intuition is that when the event a occurs the state space is changed according to the constraint φ ,

$$\begin{array}{c}
\frac{[a \rightarrow \text{Skip}] \Box \varphi \wedge [a \rightarrow \text{Skip}] [P] \Box \varphi}{[a \rightarrow P] \Box \varphi} \quad (5) \qquad \frac{\psi_{\bar{v}'} \Rightarrow \delta_{\bar{v}'_0}}{[(a \bullet \psi) \rightarrow \text{Skip}] \delta} \quad (6) \qquad \frac{\psi, F \vdash_{\text{ext}} [\text{Proc} \setminus_{A, V}] \bar{\delta} \quad \bar{\delta} \vdash \delta}{\psi \vdash [\text{Proc} \setminus_{A, V} \bullet_{\text{ext}} F] \delta} \quad (7)
\end{array}$$

Fig. 1. Some example rules from the sequent calculus; the formulae $\psi_{\bar{v}'_0}$ denotes the replacement of a variables \bar{v} in ψ with fresh variables \bar{v}_0 .

where unprimed variables in φ refer to the variable valuations before the occurrence of a and primed variables to the valuations after a . The intuition behind an undefined process like $(\text{Proc} \setminus_{\{a, b\}, \{v\}} \bullet_{DC} F)$ is that during the execution of the process arbitrary behaviour is allowed provided that the DC formula F is not violated. The events a and b are forbidden and the variable v cannot be changed in this execution. An undefined process marked with the ∞ symbol Proc^∞ will never terminate.

We embedded CSP into dynamic logic [9] to reason about CSP processes with data and undefined processes. The idea of this dynamic logic extension dCSP is to use CSP processes with data and undefined processes instead of programs within the box operator $[\cdot]$ and the diamond operator $\langle \cdot \rangle$. The dynamic logic operator $[P] \Box \varphi$ expresses that on all runs of the CSP process always φ holds, whereas $[P] \delta$ states that after every run δ is true. Analogously, $\langle P \rangle \diamond \varphi$ is used to express that there is at least one run where eventually φ holds.

To prove validity of dCSP formulae, we define a set of verification rules in a sequent-style proof calculus. Given finite sets of formulae Δ and Γ , a *sequent* $\Delta \vdash \Gamma$ is an abbreviation for the formula $\bigwedge_{\varphi \in \Delta} \varphi \Rightarrow \bigvee_{\psi \in \Gamma} \psi$. Our sequent calculus consists of rule schemata of the shape $\frac{\Phi_1 \vdash \Psi_1 \quad \dots \quad \Phi_n \vdash \Psi_n}{\Phi \vdash \Psi}$ that can be instantiated with arbitrary contexts, i.e., for every Δ and Γ the rule $\frac{\Delta, \Phi_1 \vdash \Psi_1, \Gamma \quad \dots \quad \Delta, \Phi_n \vdash \Psi_n, \Gamma}{\Delta, \Phi \vdash \Psi, \Gamma}$ is part of the calculus. As usual, formulae above the line are premises and the formula below the line the consequence: if the premises (and possibly some side-conditions) are true then the consequence also holds.

Figure 1 gives some example proof rules. The rule in (5) reduces a CSP prefix expression: to prove that $\Box \varphi$ holds for $a \rightarrow P$ we have to show that *during* execution of a $\Box \varphi$ holds and that *after* the occurrence of a during every run of P also $\Box \varphi$ holds. The following rule (6) reduces a single occurrence of an event a in a process $a \rightarrow \text{Skip}$. The idea is to symbolically execute the data change as defined in the constraint ψ of event a : after an execution of the data change in ψ the post-state of a variable v given by v' need to coincide with the pre-state of this variable in δ . Hence, to show that after every execution of $a \rightarrow \text{Skip}$ the dCSP formula δ holds, we show that the constraint ψ , where every primed variable v' is replaced by a fresh variable v_0 , implies $\delta_{\bar{v}'_0}$, i.e., δ , where every v replaced by v_0 . Rule (7) demonstrates how undefined processes with assumptions are handled. To show that on every run of a process $(\text{Proc} \setminus_{A, V} \bullet_{\text{ext}} F)$ the dCSP formula δ is valid, we need to show that a new constraint $\bar{\delta}$ is valid in the logic of F and that in our sequent calculus, $\bar{\delta}$ implies δ . If F is a DC formula then we may show $\bar{\delta}$ with existing proof methods for DC [2]. By this means, our approach flexibly integrates arbitrary timed logics to formulate assumptions on undefined processes.

$$\begin{array}{l}
 A = \{check, fail, pass, extend\}, C = \{RD, CT\} \\
 System \stackrel{c}{=} FAR \overset{o}{\circlearrowleft} check \bullet \varphi_{check} \rightarrow \quad \varphi_{check} = \Xi(sf) \wedge sf \leq RD \wedge ok' = false \\
 \quad (fail \bullet \varphi_{fail} \rightarrow REC \quad \vee \Xi(sf) \wedge sf > RD \wedge ok' = true \\
 \quad \square pass \bullet \varphi_{pass} \rightarrow System) \quad \varphi_{fail} = \Xi(sf) \wedge ok = false \\
 \quad \square extend \bullet \varphi_{extend} \rightarrow System \quad \varphi_{pass} = \Xi(sf) \wedge ok = true \\
 FAR \stackrel{c}{=} Proc_{\setminus A, C} \bullet F_{FAR} \quad F_{FAR} = \neg \diamond ([sf > RD] \wedge \ell < CT \wedge [sf \leq 0]) \\
 REC \stackrel{c}{=} Proc_{\setminus A, C}^{\infty} \bullet F_{REC} \quad F_{REC} = \neg \diamond ([sf > 0] \wedge [sf \leq 0]) \\
 \varphi_{extend} = sf' > sf
 \end{array}$$

Fig. 2. VA for a small Train Control System

4 Conclusion

Experimental Results. To validate our approach, we verified an architecture for the small example Train Control System in Fig. 2 motivated by the ETCS [1]. We were able to prove the desired safety property $sf > RD \vdash [System] \square sf > 0$ using the presented sequent calculus. To apply rules like rule (7), we made use of automatic DC verification methods [2,10]. In a second step, we proved the correctness of a concrete instantiation of the VA from Fig. 2. This instantiation were given as a COD model, for that direct verification was not possible (timeout after 80h) due to its complexity with 19 real-valued variables, over 300 locations, and 17000 transitions. But as the model is an instantiation of the VA, which can be syntactically checked with a simple refinement rule, we only needed to verify the local DC formulae F_{FAR} and F_{REC} (Fig. 2) to conclude the safety of the entire system. This was done automatically with the PEA toolkit [10] in 7h (F_{FAR}) and 4m (F_{REC}), respectively.

Related work. Our work is inspired by [11], where a fixed DC design pattern for cooperating traffic agents is introduced. Other approaches to combine CSP with data and real-time are, e.g., [7] and [12]. The former, which we also make use of in this work, is not appropriate for a proof-rule base approach because of the more complex combination that integrates CSP, DC, and OZ [13] in an object-oriented class structure. The latter likewise integrates CSP within Z constructs. Further combinations of CSP, OZ, and a real-time language are TCOZ [14] and RT-Z [15]. There is a lot of work in compositional methods for real-time systems: [16,17] introduce a sequent calculus to verify temporal properties for hybrid systems; they also examine fragments of the ETCS as case study. Compositional techniques for the verification of operationally specified real-time systems like timed automata can be found, e.g., in [18,19]. A general view on formalisation techniques for design patterns gives [20], but there, verification of real-time systems is not considered. A related approach using design patterns for a high-level real-time language is [21]: timed automata patterns for a fixed set of timing constraints are given and formally linked to TCOZ.

This is work in progress: We defined CSP with data and undefined processes, the embedding into dynamic logic, and a set of proof rules. Furthermore, the refinement rule for the instantiation of VAs with concrete COD specifications is proven correct. A proof for the correctness of the calculus is not finished yet. We

have tool support [10] for checking local DC assumptions. Tool support for our sequent calculus and the refinement rule is future work. But experiments with examples from the railway domain and automatic verification of the most time-consuming parts (checking DC assumptions) show the success of our method.

References

1. ERTMS User Group, UNISIG: ERTMS/ETCS System requirements specification. <http://www.aEIF.org/ccm/default.asp> (2002) Version 2.2.2.
2. Meyer, R., Faber, J., Hoenicke, J., Rybalchenko, A.: Model checking duration calculus: A practical approach. *Formal Aspects of Computing* **20** (2008) 481–505
3. Zhou, C., Hansen, M.R.: *Duration Calculus*. Springer (2004)
4. Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice-Hall (1985)
5. Roscoe, A.: *Theory and Practice of Concurrency*. Prentice Hall (1998)
6. Woodcock, J., Davies, J.: *Using Z - Specification, Refinement, and Proof*. Prentice Hall, London (1996)
7. Hoenicke, J.: *Combination of Processes, Data, and Time*. PhD thesis, University of Oldenburg, Germany (2006)
8. Faber, J., Jacobs, S., Sofronie-Stokkermans, V.: Verifying CSP-OZ-DC specifications with complex data types and timing parameters. In Davies, J., Gibbons, J., eds.: IFM. Volume 4591 of LNCS., Springer (2007) 233–252
9. Harel, D., Kozen, D., Tiuryn, J.: *Dynamic Logic*. Foundations of Computing. MIT Press (2000)
10. Hoenicke, J., Meyer, R., Faber, J.: PEA Toolkit. <http://csd.informatik.uni-oldenburg.de/projects/epea.html> (2006) University of Oldenburg, Germany.
11. Damm, W., Hungar, H., Olderog, E.R.: Verification of cooperating traffic agents. *International Journal of Control* **79** (2006) 395 – 421
12. Woodcock, J.C.P., Cavalcanti, A.L.C.: A concurrent language for refinement. In Butterfield, A., Pahl, C., eds.: IWF’01. BCS Elec. Works. in Computing (2001)
13. Smith, G.: *The Object Z Specification Language*. Kluwer Academic P. (2000)
14. Mahony, B.P., Dong, J.S.: Blending object-Z and timed CSP: An introduction to TCOZ. In: ICSE. (1998) 95–104
15. Sühl, C.: An overview of the integrated formalism RT-Z. *Formal Asp. Comput* **13** (2002) 94–110
16. Platzer, A., Quesel, J.D.: Logical verification and systematic parametric analysis in train control. In Egerstedt, M., Mishra, B., eds.: HSCC 2008. Volume 4981 of LNCS., Springer (2008) 646–649
17. Platzer, A.: *Differential Dynamic Logics: Automated Theorem Proving for Hybrid Systems*. PhD thesis, University of Oldenburg, Germany (2008)
18. Larsen, K.G., Pettersson, P., Yi, W.: Compositional and symbolic model-checking of real-time systems. In: Proceedings of the 16th IEEE Real-Time Systems Symposium. (1995) 76–89
19. Berendsen, J., Vaandrager, F.W.: Compositional abstraction in real-time model checking. In Cassez, F., Jard, C., eds.: FORMATS. Volume 5215 of LNCS., Springer (2008) 233–249
20. Taibi, T.: *Design Pattern Formalization Techniques*. IGI Publishing, Hershey, PA, USA (2007)
21. Dong, J.S., Hao, P., Qin, S., Sun, J., Yi, W.: Timed patterns: TCOZ to timed automata. In Davies, J., Schulte, W., Barnett, M., eds.: ICFEM 2004. Volume 3308 of LNCS., Springer (2004) 483–498