cTc – A Tool Supporting the Construction of cTLA-Specifications

Carsten Heyl*
Arnulf Mester*
Heiko Krumm*

ABSTRACT The design tool cTc has been developed to support the construction and refinement of formal specifications of distributed software systems. It concentrates on the design by stepwise refinement, where refinement steps correspond to the integration of predefined process patterns. cTc processes modules written in the compositional TLA specification style cTLA. It applies the cTLA specification operations and generates the resulting specification modules. We outline the background of cTc and give an overview on its functionality and architecture.

1 Introduction

The compositional TLA specification style cTLA and its application to the design of distributed systems have been discussed in [14] and can be summarized as follows.

cTLA is based on TLA [10] and its understanding of correct refinements [1]. It is a dialect of TLA+ [9] and tailored to the modular description of process systems. Processes perform state transitions on private variables. Process interaction is modelled by joint actions similarly to Lotos [8]. cTLA modules define generic process types. A type may describe elementary processes or subsystems where the subsystems are defined by a set of process instances and their coupling. Due to cTLA style conventions, processes and subsystems inherit their elementary safety and liveness properties to embedding systems. Thus, cTLA can supply specification operations which are based on refinement and process composition, and the application of an operation to a set of argument process types results in a new type which is a correct refinement of the arguments. The utilization of cTLA for the decompositional verification of systems is described in [4, 5].

*Universität Dortmund, Fachbereich Informatik, D-44221 Dortmund, Germany, Fax +49 231 755-4730, {heyl|mester|krumm}@ls4.informatik.uni-dortmund.de

According to our experience, the software design for distributed applications mainly deals with the consistent integration and combination of known patterns. This observation also has been made in the broader field of general software engineering leading to the non-formal approach of so-called design patterns (cf. [2]). With respect to this, our approach aims to the formalisation and tool assistance of the design pattern approach. Generic cTLA process types can describe behavioural patterns formally, e.g., distributed algorithms, communication service user scenarios, and application programming interface scenarios can be modelled by re-usable specification modules. The integration of patterns into intermediate system descriptions, the combination of patterns, and the restructuring of system descriptions can be performed by the application of cTLA specification operations.

Several example software developments showed us the suitability of our approach. Furthermore, prototypical tool implementations of the specification operations contributed to the designers' productivity as reported in [14]. Meanwhile, we refined the syntax of cTLA and enriched the set of specification operations. The tool cTc has been developed which supports the enhanced syntax and set of operations.

In the sequel we sketch the notation of the input, output, and commands of cTc by means of the cTLA syntax and specification operations. Furthermore, we address aspects of the implementation of cTc .

2 cTLA syntax and Specification Operations

The following definition of the simple process type *Prepost* gives an impression of the cTLA syntax [7].

```
PROCESS Prepost(tdata : CONST; f(a) : CONST)
BODY
  VARIABLES d : tdata;
            s : {"idle", "pre", "post"} ;
  INIT ≙    s = "idle";
  ACTIONS   Start(p : tdata) ≙ s = "idle" ∧ d' = p ∧ s'="pre" ;
            Step ≙ s = "pre" ∧ d'=f(d) ∧ s'="post" ;
  WF :      Start,Step;
END
```

The header declares the type name and introduces the generic type parameters. The body starts with the declaration of the private state variables and continues with the definitions of Init-predicate and actions. At last fairness assumptions are expressed.

Seven major specification operations exist: 1. The **compose** operation defines a process system by instantiating process patterns and the coupling of these processes. 2. The **refine** operation supports the refinement of state

variables of a pattern. 3. The integrate operation is a composition with a following refinement. 4. The split operation transforms a process into a process system. 5. The array operation introduces an array of process pattern instances. 6. The split actions operation introduces a case decision into an action. 7. The combine operation computes a proposal for a composition. The coupling is implicitly defined by the identity of action names.

As a small example we show an integrate operation application. Process *IAppl1* describes an application of integrate. The imported module *aoi* and the instantiated process type *Auxvar* are shown, *Prepost* has been described above.

```
PROCESS IAppl1
IMPORT aoi;
BODY
 VARIABLES
  id,od : aoi;
  s : {"idle", "pre", "post"} ;
 INTEGRATE
  A : Auxvar(aoi) SUBSTITUTE A.v BY id;
  P : Prepost(aoi, DoubleItems)
      SUBSTITUTE
       P.s BY s,
       P.d BY (IF s="post" THEN od
                          ELSE id);
 ACTIONS
  Start(p : aoi) ≙
    A.Write(p) ∧ P.Start(p);
  Step ≙ A.Stutter ∧ P.Step;
END
```

```
CONSTANT MODULE aoi
CONSTANTS
 aoi ≙ SET [Nat→Integer];

 double all array elements
 DoubleItems(array : aoi) ≙
  [ FCN n ∈ DOMAIN(array)
    ↦ 2*array[n] ];
END aoi

PROCESS Auxvar(tdata : CONST)
BODY
 VARIABLES
  v : tdata;
 INIT ≙ true;
 ACTIONS
  Write(p : tdata) ≙ v'=p;
 WF : Write;
END
```

The result is:

```
PROCESS IAppl1
IMPORT aoi;
BODY
  VARIABLES id,od : aoi;
            s : {"idle", "pre", "post"} ;
  INIT ≙    true ∧ s = "idle";
  ACTIONS   Start(p : aoi) ≙ id'=p ∧ s="idle"
              ∧ (IF s="post" THEN od ELSE id)'=p ∧ s'="pre";
            Step ≙ id'=id ∧ s="pre" ∧ s'="post"
              ∧ (IF s="post" THEN od ELSE id)' =
                DoubleItems((IF s="post" THEN od ELSE id));
  WF : Start,Step;
END
```

3 cTc Implementation

The main features of cTc [1] are: syntax check of cTLA specification elements and specification operations, type check of specification elements and specification operation application[2], computation of the specification operation results, transformation of cTLA specifications into tTLA+[3] specifications to facilitate the use of existing tools [13] for tTLA+ specifications (e.g. interpreter/animator, browser, model checker, otter theorem prover frontend/lemma generator), and inclusion of tTLA+ CONST modules. cTc has been designed as batch-oriented tool, which detects its desired actions from its input and command line arguments.

It has been implemented [6] in C using the GMD *cocktail* compiler toolset [3]. This toolset consists of scanner, parser, attribute evaluator and transformation generators and provides not only lex/yacc style parsing support but also supplies all routines needed to construct, check, evaluate and transform an attributed syntax tree and all routines needed to use related data structures like symbol tables. Furthermore, extensive test support is available. The scanner and parser has been realized with the *cocktail pars/rpp/lalr/rex* tools. The analysis of the static semantic relies on an attribute evaluator generated from an attributed grammar specification by the *ast* tool. Of major importance has been the tree transformation generator *puma*, which was utilized to implement the specification operations, the cTLA to tTLA+ transformation and all unparsing.

From 24K lines problem and language description *cocktail* generated 84K lines C-code. On a Sun ELC workstation (25 MHz SPARC processor) the tools analyses approx. 1000 lines of specification text per second.

4 Concluding Remarks

The modular *cocktail* descriptions for data structures, transformation rules and attribute grammars allowed for easy extension and language evolution.

Despite some small examples, an early tool prototype has been used to support a Message Transfer Agent development [11], resulting in a interoperable SMTP-conformant [16] implementation. The experiences drawn from these first case studies are encouraging. Beside other advantages, the quality of the resulting implementation is very high such that the debugging of the system has been reduced drastically.

Further work includes a tool for automated simplifications and a C-code generator for implementation-near specifications.

---

[1] Acronym for compositional TLA construction.

[2] Based on an extension of the type checking used in a tTLA+ browser [15].

[3] tTLA+ [12] is a tool oriented notation of TLA+ [9].

5 REFERENCES

[1] M. Abadi, L. Lamport: The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns. Elements of Reusable Object-Oriented Software. 1994

[3] J. Grosch, H. Emmelmann: A Tool Box for Compiler Construction. Compiler Generation Report No. 20, GMD Karlsruhe, Jan 1990.

[4] P. Herrmann, H. Krumm: Compositional Specification and Verification of High-Speed Transfer Protocols. In: S.T. Vuong and S.T. Chanson (Eds.) *Protocol Specification, Testing, and Verification XIV*. 1994.

[5] P. Herrmann, H. Krumm: Re-Usable Verification Elements for High-Speed Transfer Protocol Configurations. In: P. Dembinski and M. Sredniawa (Eds.) *Protocol Specification, Testing, and Verification XV*, 1995.

[6] C. Heyl: Tool Support for the Compositional Design of Distributed Systems in TLA. Diploma Thesis (in German), Univ. Dortmund, Informatik IV, 1995.

[7] C. Heyl, A. Mester: cTLA syntax definition. Univ. Dortmund, Informatik IV. Technical Report RvS-TLA-95/49, Oct 1995.

[8] ISO: *LOTOS: Language for the temporal ordering specification of observational behaviour*. International Standard ISO/IS 8807, 1987.

[9] L. Lamport: TLA^+: syntax and semantics. Digital Equipment Corporation, Systems Research Center, Preliminary Version, Feb 1992.

[10] L. Lamport: The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[11] K. Luttmann: Formal Design and Implementation of an Internet Mail System. Diploma Thesis (in German), Univ. Dortmund, Informatik IV, 1996.

[12] A. Mester: tTLA+ 1 syntax description. Univ. Dortmund, Informatik IV, Technical Report RvS-TLA-92/4, Jan 1993.

[13] A. Mester, P. Herrmann: Tools for TLA-based Specifications. Univ. Dortmund, Informatik IV, Technical Report RvS-TLA-94/35, 1994.

[14] A. Mester, H. Krumm: Composition and Refinement Mapping based Construction of Distributed Applications. In: Uffe H. Engberg, Kim G. Larsen, Arne Skou (Eds.): *Tools and Algorithms for the Construction and Analysis of Systems*, BRICS NS-95-2, 290–303, Aarhus, Denmark, May 1995

[15] O. Meier: Tool Support for getting into TLA Specifications of Distributed Systems. Diploma Thesis (in German), Univ. Dortmund, Informatik IV, 1994.

[16] J.B. Postel: Simple Mail Transfer Protocol. Request for Comments 821, August 1982