http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

Information Technology Journal 7 (4): 623-630, 2008
ISSN 1812-5638
© 2008 Asian Network for Scientific Information

Combination of Model Checking and Theorem Proving to Develop and Verify Embedded Software

Zhong Xu and Xiao Jianyu
Department of Computer, Changsha University, Changsha Hunan, 410003, China

Abstract: A strategy of combining Model checking and theorem proving techniques to develop and verify high confidence embedded software was proposed. First, the UML state machine of software model was transformed into the input modeling language of model checker MOCHA and the model was analyzed in the model checking tool with associated property specifications expressed in temporal logic. The model checker can give counterexamples if it decided that the model can not satisfy any of the specified properties, which can be used to modify the software's UML design model. The UML model which has been verified by MOCHA was then transformed into abstract specifications of theorem prover Atelier-B, in which the model would be refined, verified and translated into source C code. The transformation rules from UML state machine to REACTIVE MODULES and to B abstract machines were given. The experiment showed that the proposed strategy can effectively improve the development and verification of high-confidence embedded software.

Key words: Model checking, theorem proving, high-confidence software, software verification, UML

INTRODUCTION

Verification of embedded software has long been a concerned difficult problem, especially those software embedded in the industrial control devices, astronautical aircrafts or nuclear power facilities. Formal methods have attracted more and more attention in the development and verification of this kind of high confidence software due to its rigor and precision, which is based on mathematics (Singhal, 2001; Steven, 2003; Farokh, 2001; Heimdahl and Heitmeyer, 1998).

Formal methods are divided into two categories: formal specification and formal verification. Formal verification is based on formal specification and is used to decide whether a checked system supports some given properties expressed in temporal logic. There are two approaches to formal verification: model checking (Clarke *et al*., 1999) and theorem proving (Zhang, 1997). In model checking, the system's behavior is represented by a labelled state-transition system (abbreviated 'S'); the system's properties are described by modal/temporal logic formula (abbreviated 'F') and thus, the question "whether the system satisfies the expected property ?" is translated into another mathematical question "Is the state-transition system 'S' a model of formula 'F' ? " (abbreviated 'S |= F ?'), which can be decided for a finite state system by means of exhaustively searching the system's state space. In theorem proving, software system and its properties are all specified by logics. Whether the system satisfies the expected properties is decided by a proving process in a formal system based on axioms and associated inference rules, just as the theorem proving in mathematics. Specially, it is the software system that forms the axiomatic system. Model checking and theorem proving are complementary. The strong points of model checking include high degree of automation and its ability of providing counterexamples when it decided that the system violates some properties. The main hurdle of model checking is the problem of state explosion and for software, the currently existed model checking tools can only validate the design model and none of them can contribute to the implementation of software. Theorem proving can handle infinite state space based on induction in infinite domain and many tools of theorem proving support automatic generation of source code. Its shortcomings include low degree of automation, necessity of manual intervention during proof and unability to provide readable counterexample when fail in proof.

A recent trend of formal verification is to combine model checking and theorem proving techniques (Shankar, 2001; Manna *et al*., 2003; Mikhailov and Butler, 2002; Berezin, 2002; McMillan, 1999). In Shankar (2001), the theorem prover PVS was enhanced with tools for abstraction and model checking. In Manna *et al.* (2003), model checker SteP was integrated into an automatic deductive theorem prover. In Mikhailov and Butler (2002), B theorem prover and Alloy model checker was combined.

Corresponding Author: Zhong Xu, Department of Computer, Changsha University, Changsha Hunan, 410003, China

All of these work are directed against the problem of state explosion and to our knowledge, little work has been reported to make use of the functional complement of the two techniques in practice of high confidence software engineering.

This study proposed a suit of practical scheme to verify embedded software, combining model checking tool MOCHA (de Alfaro *et al.*, 2000) and theorem proving tool Atelier-B (ClearSy, 1998) which is not only a theorem prover but also a powerful formal software development environment with many facilities. The programme of the scheme is: i) Transform the UML state machine of software design model into MOCHA's input language REACTIVE MODULES and verify the satisfiability of expected properties in MOCHA; ii) Transform the already verified UML model into abstract specifications of B language (Abrial, 1996) and refine it into implementation model described by B0 language step by step; iii) Generate source C code by facilities of Atelier-B. Experiments show that the proposed scheme can improve the development and validation of embedded software with little manual intervention.

MOCHA AND B METHOD

Introduction of model checker MOCHA: MOCHA is an interactive model checking environment developed by University of California, Berkely. It is used for concurrent system's specification, simulation and verification. The main facilities supported by MOCHA include: i) Modeling language reactive modules which is used to formally specify hardware or software systems, supporting heterogeneous modeling framework and hierarchical design at different levels of abstraction; ii) Simulating environment Simulator with an interactive graphical user-interface to simulate modules which supports system execution by randomized or manual trace generation; iii) Invariant Checkers with enumerative and symbolic search engine which can check state invariants and transition invariants and can provide counterexamples when the checking fails. Details of MOCHA was described by L. de Alfaro *et al.* (2000).

Introduction of B method: B stands for a methodology, a language and a toolset for the specification, design and coding of software systems introduced in (Abrial, 1996, 2003). B is based on viewing a program as a mathematical model and the concepts of pre- and postconditions, of non-determinism and weakest precondition. The B language is based on the concept of abstract machines. An abstract machine consists of variables and operations and is defined as follows:

```
MACHINE M( … )
  CONSTANTS
    …
  VARIABLES
    …
  INVARIANT
    …
  INITIALIZATION
    …
  OPERATIONS
    …
```

The state of the variables is the state of the machine bound by the INVARIANT. Operations may change the machine's state with respect to the invariant. An Abstract Machine is refined by reducing non-determinism and abstract functions until a deterministic implementation is reached which may be translated into executable code.

The B method is based on the notion of refinement of specifications supported by the B language. Refinement means the replacement of a machine M by a machine R and that the operations of M are defined by R with identical signatures. Refinement can be conducted in three different forms: i) the removal of the pseudo-code (precondition and choice); ii) the introduction of the classical control structures of programming (sequencing and loop) and iii) the transformation of the data structures (sets, relations, functions, sequences and trees). A final refinement defines the implementation of the system which is denoted by B0 language. From B0, the B toolset can automatically generate Ada, C, or C++.

Along the process of specification and refinement in the B method, the mechanism of B generates proof obligations for consistency verification automatically. The theorem prover in B toolset (B-Toolkit or Atelier-B) can prove much of the proof obligations automatically. As for the non-obvious proof obligations, manual intervention is needed to input lemmas to help the theorem prover.

PROGRAMME OF EMBEDDED SOFTWARE'S VERIFICATION, REFINEMENT AND CODE GENERATION

The programme of software's development and verification of our scheme is shown in Fig. 1. In the scheme the UML state machine of software system's design model is the startpoint and focus of the whole process. The concrete steps of development is described as follows:

- The UML state machine of software model is transformed into MOCHA's REACTIVE MODULES description and the execution of the software is simulated in MOCHA's Simulator. The properties of

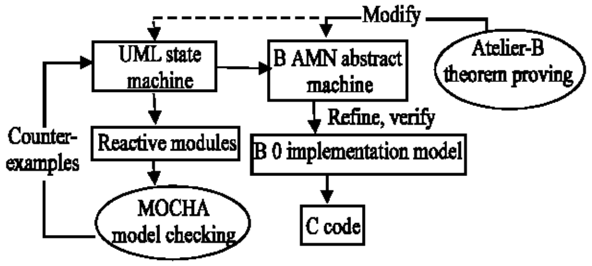


Fig. 1: Flow of embedded software's development and verification

system are expressed as state invariants or transition invariants and are verified through invariants checking. MOCHA would provide detail path of counterexamples if it finds some properties has been violated. These valuable counterexamples can be used to modify the UML state machine and the invariants checking will be repeated on the modified UML model. The verification-modification process should be repeated over and over until the invariants checking of the UML state machine succeeds. This step can find out faults in the software's design model and rule them out as early as possible. It can reduce the possibility of manual interventions in the next step of software model's consistency verification in B's theorem prover.

- The already verified UML state machine is transformed into B AMN specification and is verified in Atelier-B. If there are any proof obligations that can not be proved, the B AMN abstract model will be modified and re-verified until all the proof obligations have been accomplished. If the B AMN model can not get through Atelier-B's consistency verification assuredly, the initial UML state machine might be re-designed. But then, the step i) should be redone.
- The verified B AMN abstract model is refined progressively into implementation model in B0 language in Atelier-B. The correctness of the refinement is guaranteed by Atelier-B's refinement verification mechanism mentioned earlier.
- The implementation model of the software is translated into C code by Atelier-B automatically.

The difficult points of the above process include: i) how to describe UML state machine in modeling language REACTIVE MODULES; ii) how to describe UML state machine in B AMN. The following is our solutions to the problems.

Transformation rules from UML state machine to reactive modules: UML state machine and MOCHA's modeling language REACTIVE MODULES are all designed to describe concurrent systems. They have similar semantics. The transformation rules from UML concurrent state diagram which is often used to describe the design scheme of embedded software are shown in Table 1.

Transformation rules from UML state machine to B AMN: B AMN can also describe concurrent systems. The transformation rules from UML concurrent state diagram to B AMN are shown in Table 2.

Experiment and analysis: We have developed and verified a control software embedded in a mechanical manipulator, using the tools MOCHA and Atelier-B.

The system which is shown in Fig. 2 has three control components: device A which is for holding goods, device E which is the target place for holding goods and manipulator T for transporting goods from A to E. Each device can place only one piece of goods. The constraints of the system's function include: i) goods can be placed on device A only when A is idle (represented as vid); ii) goods can be uploaded on manipulator T only when T is idle and at the same time device A is occupied (represented as occ); iii) goods can be downloaded only when device E is idle; iv) goods can be downloaded only when T positions high (represented as high); v) goods can be uploaded only when T positions low (represented as low); vi) T can ascend only when it is occupied; vii) T can descend only when it is idle.

The experiment was conducted as follows:

- The primitive UML state machine of the control software system was designed according to requirements (because there are many variants of primitive design models and there are many flaws in them, the primitive models are not shown here.);
- The primitive UML state machine of the system was transformed into B AMN abstract model and whose consistency was verified in Atelier-B. The result of the proof obligations was shown in Table 3, where the referenced parameters are somewhat the same as in Fig. 3 and the obvious proof obligations column lists the amount of proof obligations which can be accomplished by Atelier-B automatically and the non-obvious proof obligations column lists the amount of proof obligations which cannot be accomplished by Atelier-B at present.

Table 1: Transformation rules from UML concurrent state diagram to reactive modules

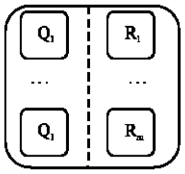
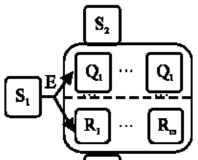
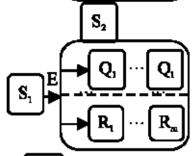
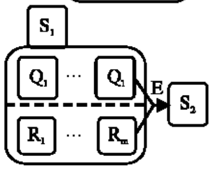
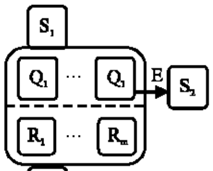
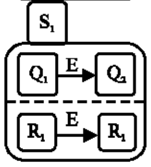
UML concurrent state diagram	Reactive modules description	Notes
	MODULE S is Q||R MODULE Q $q_1 : \{Q_1.q_1, Q_2.q_1, \ldots, Q_l.q_1\} \ldots$ $q_n : \{Q_1.q_n, Q_2.q_n, \ldots, Q_l.q_n\}$ MODULE R $r_1 : \{R_1.r_1, R_2.r_1, \ldots, R_m.r_1\} \ldots$ $r_N : \{R_1.r_N, R_2.r_N, \ldots, R_m.r_N\}$	The concurrent sub-states of UML concurrent state diagram are declared as Modules in REACTIVE MODULES. The relativity of substrates is embodied in relativity among variables in Modules.
 	MODULE Q … Update … $[]x_1 = s_1.x_1 \wedge \ldots \wedge x_M = s_1.x_m$ $\rightarrow q_1' := Q_1.q_1; \ldots; q_n' := Q_1.q_n$ MODULE R … Update … $[]x_1 = s_1.x_1 \wedge \ldots \wedge x_M = s_1.x_m$ $\rightarrow r_1' := R_1.r_1; \ldots; r_N' := R_1.r_N$	When a transition enters into an UML concurrent compound state, the number of control lines will increase and enter into each sub-states branchingly. In REACTIVE MODULES, all the related concurrent sub-state variables should be updated. If the arrow of transition only points to the sounding line of the super-state, each of the concurrent sub-states in the surrounding line should have an initial state. The corresponding REACTIVE MODULES description is the same as in the situation of no surrounding line.
	MODULE S … update … $[]q_1 = Q_1.q_1 \wedge \ldots \wedge q_n = Q_1.q_n$ $\wedge r_1 = R_1.r_1 \wedge \ldots \wedge r_N = R_m.r_N$ $\rightarrow x_1' = s_2.x_1; \ldots; x_M' = s_2.x_M$	When a transition leaves an UML concurrent compound state, it should leave from all the sub-states and the number of control lines will decrease to 1.
	MODULE S … update $[]q_1 = Q_1.q_1 \wedge \ldots \wedge q_n = Q_1.q_n$ $\rightarrow x_1' = s_2.x_1; \ldots; x_M' = s_2.x_M$	When a transition leaves an UML concurrent compound state, it would be seen as leave from the whole concurrent sub-states if the arrow begins from one of the concurrent sub-states.
	MODULE Q … Update … $[]q_1 = Q_1.q_1 \wedge \ldots \wedge q_n = Q_1.q_n$ $\rightarrow q_1' := Q_2.q_1; \ldots; q_n' := Q_2.q_n$ MODULE R … Update … $[]r_1 = R_1.r_1 \wedge \ldots \wedge r_N = R_1.r_N$ $\rightarrow r_1' = R_2.r_1; \ldots; r_N' = R_2.r_N$	One event may cause transitions from two concurrent sub-states.

Table 2: Transformation rules from UML concurrent state diagram to B AMN

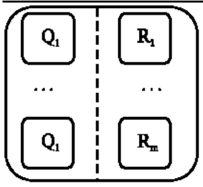
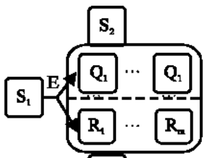
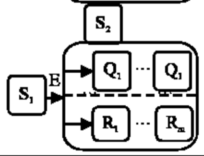
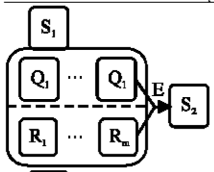
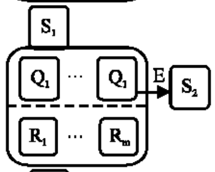
UML concurrent state diagram	AMN description	Notes
	SETS $S = \{S_1, \ldots, S_n\}$; $Q = \{Q_1, \ldots, Q_l\}$; $R = \{R_1, \ldots, R_m\}$ VARIABLES s, q, r INVARIANT $s \in S \wedge q \in Q \wedge r \in R$	The concurrent sub-states of UML concurrent diagram are declared as variables in AMN. When the concurrent sub-states belong to the same super-state, the corresponding variable will be associated.
 	EVENTS E A IF $s = S_1$ THEN $s := S_2 \parallel q := Q_1 \parallel r := R_1$ END	When a transition enters into an UML concurrent compound state, the number of control lines will increase and enter into each sub-states branchingly. In AMN, all the corresponding concurrent sub-state variables should be set. If the arrow of transition only points to the surrounding line of the concurrent super-state, each of the concurrent sub-states in the surrounding line should have an initial state and the AMN description is the same as the branching situation.

Table 2: Continued

UML concurrent state diagram	AMN description	Notes

	EVENTS
	E A
	IF $s = S_1 \wedge q \wedge = Q_1 \wedge r = R_m$
	THEN $s := S_2$
	END
		When a transition leaves an UML concurrent compound state, it should leave from all the sub-states and the number of control lines will decrease to 1. In AMN, the concurrent sub-states variables should be tested in the guarding conditions of corresponding events.

	EVENTS
	EA
	IF $s = S_1 \wedge q = Q_1$
	THEN $s := S_2$
	END
		When a transition leaves an UML concurrent compound state, it would be seen as leave from the whole concurrent sub-states if the arrow begins from one of the concurrent sub-states. In AMN, a new value should be assigned to the super-state variable.

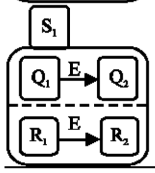
	EVENTS E A BEGIN IF $q = Q_1$ THEN $q := Q_2$ END || IF $r = R_1$ THEN $r := R_2$ END END	One event may cause transitions from two concurrent sub-states. In AMN, this situation can be described by two concurrent event clauses.

Table 3: Proof obligations of the primitive model's B specification

Modules	Obvious proof obligations	Non-obvious proof obligations
T	138	120
A	26	13
E	430	430
PostT	18	10

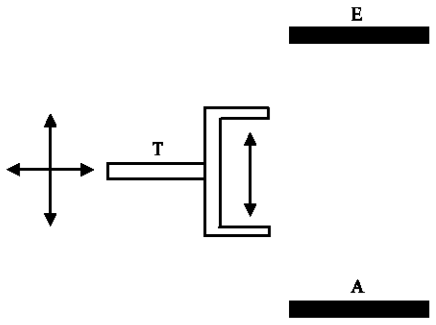
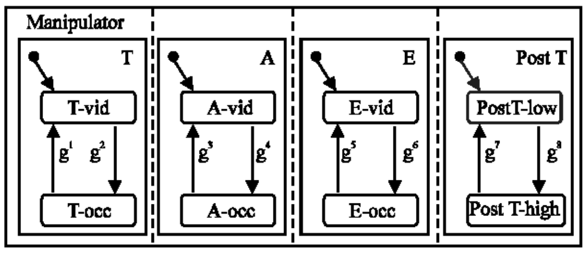


Fig. 2: The manipulator control system



g^1: dcht [In E-vid & Post T-high], g^2: chgt [In A-occ & PostT-low],
g^3: chgt [In T-vid & PostT-low], g^4: arr, g^5: evac, g^6: dchgt [In T-occ & PostT-high],
g^7: descend [In T-vid], g^8: ascend [In T-occ]

Fig. 3: The last version of UML state machine of the manipulator control software

- The UML state machine model was processed according to the flow described in section 2 and the last version of the modified UML model was shown in Fig. 3 with the corresponding reactive modules description shown in Fig. 4 and the B AMN abstract modle shown in Fig. 5. After many steps of refinement in Atelier-B, the resulted implementation model was produced which is illustrated in Fig. 6. All of the two B models proof obligations were proved by Atelier-B's theorem prover with no manual intervention.

The experiment showed that the primitive UML model designed according to system's requirements always had many flaws. When it was being verified in Atelier-B, the proof obligations which can not be automatically proved by theorem prover were so many that the manual intervention was too much to be practiced. On the other side, the validated UML model after being processed by model checking tool can be easily proved in theorem prover with almost no manual help.

```
Const high, low, occ, vid;
Type StateType is {occ, vid}; type PositionType is {high, low
Module A is
External ready1: event; manipulator: StateType; position: PositionType
Interface source: StateType
Atom sourcehould controls source reads source, manipulatore, position, ready1
Init
   []true->source':=vid
update
   []source=occ&manipulator=vid&position=low->source'=vid
   []source=vid&ready1?->source':=occ

Module E is
External ready2: event; manipulator: StateType; position: PositionType
Interface: target: StateType
Atom targethold controls target reads target, manipulator, position, ready2
Init
  []true->target':=vid
Update
  []target=occ&ready2?->target':=vid
  []target=vid&manipulator=occ&postion=low->target':=occ

Module T is
External source, target: StateType; position:PositionType
Interface manipulator: StateType
Atom transport1 controls manipulator reads manipulator, source, target, position
Init
  []true->manipulator':=vid
Update
  []manipulator=occ&target=vid&position=high->manipulator:=vid
  []manipulator=vid&source=occ&position=low->manipulator:=occ

Module PostT is
External manipulator: StateType
Interface position: PositionType
Atom transtort2 controls position reads position, manipulator
Init
  []true->position':=low
Update
  []position=high&manipulator=vid->position':=low
  []position=low&manipulator=occ->position':=high
```

Fig. 4: The reactive modules description of UML state machine of the manipulator control software

```
MACHINE   Manipulator 0
VARIABLES T0
INVARIANT
 T0 : {vid, occ}
INITIALIZATION
  T0 := vid,
EVENTS
  chgt A SELECT T0 = vid THEN T0 := occ END
  dchgt A SELECT T0 = occ THEN T0 := vid END
END
```

Fig. 5: The B abstract model of the UML state machine in Fig. 3

```
MACHINE Manipulator REFINES Manipulator2
VARIABLE   T, A, E, Post T
INVARIANT
  T = T2 and  A = A3, E = E2 and  Post T : {low, high}
INITIALIZATION
  T, A, E, Post T := vid, vid, vid low
EVENTS
  chgt SELECT T = vid and A = occ and Post T = low THEN T, A:= occ, vid  END
  dchgt  SELECT T = occ and E = vid and  Post T = high THEN T, A:= vid, occ, END
  arr ASELECT A = vid THEN A := occ END
 evac A SELECT E = occ THEN E := vid END
 accend A SELECT E := occ and Post T = low THEN Post := high END
descend A SELECT E = vid and Post T = high THEN Post T := lowEND
END
```

Fig. 6: The B0 implementation model of the UML state machine in Fig. 3

CONCLUSIONS

A practical strategy of developing and verifying embedded software combining model checking and theorem proving techniques was proposed. First, UML state machine of software model was transformed into the input language of model checker MOCHA and was checked in it. The model checker can give counterexamples if it finds some flaw in the model being checked according to some given properties, which can be used to modify the software's UML design model. The already verified UML model was then transformed into B abstract model and can be refined, verified and translated into source C code in the Atelier-B environment. Experiment showed that the strategy can effectively improve the development and verification of embedded software.

The limits of our strategy include: i) the transformation from UML to MOCHA model or to B model was currently conducted manually, which may not effective in large scale system; ii) due to the fact that model checking can only handle finite state space, in our strategy, it is assumed that the variables' domain of the software model was finite. Fortunately, most of the device controlling software's state space is finite.

REFERENCES

Abrial, J.R., 1996. The B-Book-Assigning Programs to Meanings. Cambridge University Press, New York, USA.

Abrial, J.R., 2003. B#: Toward a synthesis between z and b. 3rd International Conference of B and Z Users-ZB 2003, Turku, Finland. Lectures Notes in Computer Science. Springer Verlag June 2003, pp: 168-177.

Berezin, S., 2002. Model Checking and Theorem Proving: A Unified Framework. Ph.D Thesis, Carnegie Mellon University.

Clarke, E., O. Grumberg and D. Peled, 1999. Model Checking. MIT Press.

ClearSy, B., 1998. Language Reference Manual (Version 1.8.5).

De Alfaro, L., R. Alur and R. Grosu *et al.*, 2000. Mocha: Exploiting Modularity in Model Checking, http://www-cad.eecs.berkeley.edu/˜mocha, 2000.

Farokh, B.B., 2001. High-Quality Customizable Embedded Software from COTS Components. IEEE Computer Society 20th Symposium on Reliable Distributed Systems 10-13 April. New Orleans, LA, USA., pp: 174-175.

Heimdahl, M.P. and C.L. Heitmeyer, 1998. Formal methods for developing high assurance computer systems: Working group report. 2nd IEEE Workshop on Industrial-Strength Formal Techniques, pp: 1-60.

McMillan, K.L., 1999. Circular compositional reasoning about liveness. Technical report, Cadence Berkeley Labs, Cadence Design Systems.

Mikhailov, L. and M. Butler, ZB: 2002. An Approach to Combining B and Alloy. In: Formal Specification and Development in Z and B, Bert, D. *et al.* (Eds.). ZB'2002, Springer-Verlag, Grenoble, France, pp: 140-161.

Shankar, N., 2001. Combining theorem proving and model checking through symbolic analysis. 11th Concur, LNCS 1877, pp: 1-16.

Singhal, M., 2001. Research in high-confidence distributed information. IEEE Computer Society 20th Symposium on Reliable Distributed Systems. New Orleans, LA, USA., pp: 76-79.

Steven, D.J., 2003. Formal methods in embedded design. Computer, 36 (36): 104-106.

Manna, Z., E. Chang and A. Anuchitanukul *et al.*, 2003. STeP: the Stanford Temporal Prover. Technical Report STAN-CS-TR-03-1518, Stanford University.

Zhang, H., 1997. SATO: An efficient propositional prover. In: Proceedings of the International Conference on Automated Deduction, July 1997, pp: 272-275.