UDC 519.7

Deriving homing sequences for Finite State Machines with timed guards

Tvardovskii Aleksandr (National Research Tomsk State University),
Yevtushenko Nina (Ivannikov Institute for System Programming of the RAS, National Research University Higher School of Economics)

State identification is the well-known problem in the theory of Finite State Machines (FSM) where homing sequences (HS) are used for the identification of a current FSM state, and this fact is widely used in the area of software testing and verification. For various kinds of FSMs, there exist sufficient and necessary conditions for the existence of preset and adaptive HS and algorithms for their derivation. Nowadays timed aspects become very important for hardware and software systems. In this work, we address the problem of checking the existence and derivation of homing sequences for FSMs with timed guards. The investigation is based on the FSM abstraction of a Timed FSM.

Keywords: *Finite State Machine, Timed guards, FSM abstraction, Homing sequence.*

1. Introduction

Testing is an important part of the hardware and software life cycle and since the complexity of telecommunication and other control systems permanently increases, formal methods for deriving high quality tests are in a great demand [1-2]. When deriving tests with guaranteed fault coverage of reasonable length, state identification sequences for Finite State Machines (FSM) are widely utilized [3]. Homing sequences (HS) allow determining a current state of an FSM under test and can be efficiently used for reducing testing efforts in active and passive testing [4, 5]. Various approaches for deriving homing sequences are developed and these sequences can be preset or adaptive [7, 8, 9]. Preset input sequences are derived before starting the identification procedure based on a successor tree of an FSM under investigation [6, 7] and such techniques exist for deterministic and nondeterministic, partial and complete, weakly initialized and non-initialized FSMs [8].

Nowadays time aspects become very important for digital and hybrid systems, and, respectively, classical FSMs have been extended with time variables [11-15]. A timed FSM (TFSM) is an FSM annotated with a *clock* and extended by input/output timeouts [12, 14] and input/output timed guards [11, 15]. Input timed guards describe the behavior at a given state for inputs which arrive during an appropriate time interval until the state timeout expires. As mentioned above, methods for deriving preset HS are well studied for classical FSMs and in this work, we consider the problem of the HS derivation for FSMs with timed guards.

The rest of the paper has the following structure. Section 2 contains preliminaries. In Section 3, the problem of deriving a HS for an FSM with timed guards is investigated. Section 4 contains optimization methods for solving a HS problem for FSMs with timed guards. Section 5 concludes the paper.

2. Preliminaries

In this section, we briefly remind the notions of classical and Timed Finite State Machines and discuss existing methods for representing a Timed FSM by the corresponding abstraction that is a classical FSM.

2.1. Finite State Machines

Finite State Machines (FSM) [3] or simply *machines* are used for describing the behavior of a system that moves from state to state under input stimuli and produces a prescribed output response. Formally, an FSM is a 4-tuple $\mathsf{S} = (S, I, O, h_S)$ where S is a finite non-empty set of states, I and O are input and output alphabets, and $h_S \subseteq (S \times I \times O \times S)$ is the transition (behavior) relation. Such FSMs are sometimes called non-initialized FSMs; an *initialized* FSM has the designated initial state s_0 and is a 5-tuple (S, s_0, I, O, h_S). A transition (s, i, o, s') describes the situation when an input i is applied to S at the current state s. In this case, the FSM moves to state s' and produces the output (response) o. In this work we consider complete observable machines, i.e., machines where for each pair $(s, i) \in S \times I$ there exists $(o, s') \in O \times S$ such that $(s, i, o, s') \in h_S$ and for every two transitions $(s, i, o, s_1), (s, i, o, s_2) \in h_S$ it holds that $s_1 = s_2$. A complete FSM S is *deterministic* if for each pair $(s, i) \in S \times I$ there exists exactly one $(o, s') \in O \times S$ such that $(s, i, o, s') \in h_S$.

A trace or an *Input/Output sequence* α/γ of the complete FSM S at state s is a sequence of consecutive input/output pairs starting at the state s, where α is the input sequence and γ is the corresponding output sequence. Given a complete observable FSM S, states s and p are *equivalent* if the sets of output responses at these states coincide for each input sequence. A complete deterministic FSM S is *reduced* if every two different states $s_1, s_2 \in S$ are not equivalent. FSM S is *strongly connected* if for each pair of states $s_1, s_2 \in S$ there exists a trace that takes the FSM from state s_1 to state s_2.

2.2. Timed Finite State Machines

In this paper, we consider FSMs with timed guards (TFSM), i.e., FSMs which are enriched with a clock variable and timed guards [11, 13, 15]. A non-initialized TFSM is a 4-tuple $\mathsf{S} = (I, S, O, h_\mathsf{S})$ where S is a finite non-empty set of states, I and O are input and output alphabets, Π is a set of *input timed guards* and $h_\mathsf{S} \subseteq S \times I \times O \times S \times \Pi$ is the *transition relation*. An initialized TFSM has the designated initial state s_0. An input timed guard $g \in \Pi$ describes the time domain when a transition can be executed and is given in the form of interval $<$*min*, *max*$>$ of $[0; \infty)$, where $< \in \{(, [\}$, $> \in \{),]\}$. We also denote B_S the largest finite boundary of timed guards of S. The transition $(s, i, o, s', g) \in S \times I \times O \times S \times \Pi$ means that TFSM S being at state s accepts an input i applied at time $t \in g$ measured from the initial moment or from the moment when TFSM S has produced the last output; the clock then is set to zero and S produces output o. Given TFSM S, S is a *complete* TFSM if every input is defined at every state and the union of all input timed guards at any state s under every input i equals $[0; \infty)$. A TFSM S is *deterministic* if for every two transitions (s, i, o_1, s_1, g_1), $(s, i, o_2, s_2, g_2) \in h_\mathsf{S}$, $s_1 \neq s_2$ or $o_1 \neq o_2$, it holds that $g_1 \cap g_2 = \varnothing$, otherwise, TFSM S is *nondeterministic*. A TFSM is *observable* if for every two transitions (s, i, o, s_1, g_1), $(s, i, o, s_2, g_2) \in h_\mathsf{S}$, where $g_1 \cap g_2 \neq \varnothing$, it holds that $s_1 = s_2$.

A *timed input* is a pair (i, t) where $i \in I$ and t is a real; a timed input (i, t) means that input i is applied to the TFSM at time instance t measured from the initial moment or from the moment when the last input was applied to TFSM S. A sequence of timed inputs $\alpha = (i_1, t_1) \ldots (i_n, t_n)$ is a *timed input sequence*. Given a timed input sequence $(i_1, t_1) \ldots (i_n, t_n)$, an input i_1 is applied when the clock value is equal to t_1; after applying the input, the machine produces a prescribed output and the clock is set to 0. The machine is then waiting for the next input i_2 that is applied when the clock value equals t_2. A sequence $\alpha/\gamma = (i_1, t_1)/o_1 \ldots (i_n, t_n)/o_n$ of consecutive pairs of timed inputs and outputs starting at the state s is a *timed trace* of TFSM S at state s. Similar to FSMs, α is an applied timed input sequence while γ is the corresponding output response of the TFSM to sequence α of applied inputs. For example, when the timed input $(i_1, 1.7)$ is applied to TFSM S (Figure 1) at state s_1 the TFSM moves to state s_3, produces output o_2, reset the clock and waits for the next input.

The notions of reduced and strongly connected TFSMs are similar to those of classical FSMs up to the replacement of an input sequence to a timed input sequence.

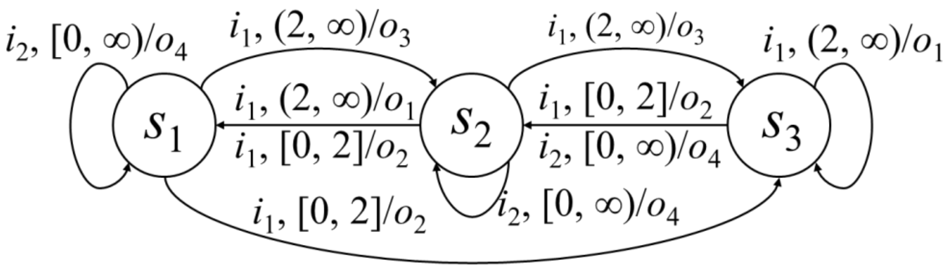

Fig. 1. FSM with timed guards S.

2.3. FSM Abstraction

In a number of cases, the behavior of a TFSM can be adequately described by a classical FSM that is called the *FSM abstraction* of the TFSM [13]. Given a complete observable possibly nondeterministic TFSM S = (S, I, O, h_S) with the largest finite boundary of timed guards B_S, a corresponding FSM abstraction $A_S = (S, I_A, O, h_{AS})$, where $I_A = \{(i, [0, 0]), (i, (0, 1)), \dots, (i, (B_S - 1, B_S)), (i, [B_S, B_S]), (i, (B_S, \infty)): i \in I\}$, can be derived as follows. There is a transition $(s, (i, g_i), o, s') \in h_{AS}$ if and only if there is a transition $(s, i, o, s', g) \in h_S$ such that $g_i \subseteq g$. For the nondeterministic TFSM S in Figure 1, Table 1 represents the flow table of the corresponding FSM abstraction where rows correspond to inputs, columns correspond to states and a corresponding item for state s and input i contains the corresponding pairs of state s' and output o such that $(s, i, o, s') \in h_{AS}$. A timed input sequence $\alpha = (i_1, t_1) \dots (i_n, t_n)$ of the FSM with timed guards S can be transferred into a corresponding sequence of inputs $\alpha_{FSM} = (i_1, g_1) \dots (i_n, g_n)$ for the FSM abstraction A_S and vice versa where $t_j \in g_j$, $j = 1\dots n$. By direct inspection, a reader can assure that for every other input, a corresponding row coincides with one of three inputs colored in grey.

Table 1. Flow table of the FSM abstraction of the TFSM in Fig 1.

i/s	s_1	s_2	s_3
$(i_1, [0, 0])$	s_3/o_2	s_1/o_2	s_2/o_2
$(i_1, (0, 1))$	s_3/o_2	s_1/o_2	s_2/o_2
$(i_1, [1, 1])$	s_3/o_2	s_1/o_2	s_2/o_2
$(i_1, (1, 2))$	s_3/o_2	s_1/o_2	s_2/o_2
$(i_1, [2, 2])$	s_3/o_2	s_1/o_2	s_2/o_2
$(i_1, (2, \infty))$	s_2/o_3	s_1/o_1, s_3/o_3	s_3/o_1

i/s	s_1	s_2	s_3
$(i_2, [0, 0])$	s_1/o_4	s_2/o_4	s_2/o_4
$(i_2, (0, 1))$	s_1/o_4	s_2/o_4	s_2/o_4
$(i_2, [1, 1])$	s_1/o_4	s_2/o_4	s_2/o_4
$(i_2, (1, 2))$	s_1/o_4	s_2/o_4	s_2/o_4
$(i_2, [2, 2])$	s_1/o_4	s_2/o_4	s_2/o_4
$(i_2, (2, \infty))$	s_1/o_4	s_2/o_4	s_2/o_4

Note that the FSM abstraction of a complete deterministic (nondeterministic) TFSM is a complete deterministic (nondeterministic) FSM and similar to the statement proven in [13] for initialized TFSMs, the following statement holds for a non-initialized TFSM.

Proposition 1. There exists a timed trace α/γ at state s of a non-initialized TFSM S if and only if the FSM abstraction $\mathsf{A_S}$ has a trace α_{FSM}/γ at state s.

3. Homing sequences for TFSMs

A Homing Sequence (HS) allows to determine a state reached by an FSM after applying this input sequence and observing the produced outputs. In this section, we define a Homing Sequence for a complete FSM with timed guards and show how this sequence can be derived based on the FSM abstraction of the TFSM.

Given a trace α/γ of a complete observable possibly nondeterministic FSM, state s' is the α/γ-successor of state s in FSM S if S moves from state s to state s' by trace α/γ. The α/γ-successor of a subset S' of states is the union of α/γ-successors over all states $s \in S'$; note that the α/γ-successor of a state s as well as of a subset S' can be the empty set. In the same way the α/γ-successor is defined for a timed trace α/γ for a complete observable TFSM.

An input sequence α is a *Homing Sequence* (HS) for a non-initialized complete observable possibly nondeterministic FSM S if for each output sequence γ, the α/γ-successor of S is a singleton or does not exist. A HS for a complete observable FSM can be derived using an appropriate truncated successor tree [3, 8, 16].

When timed FSMs are considered, the value of the clock variable must be taken into account before starting a homing experiment. By definition, the value of the TFSM clock is always reset to zero when an input is applied and respectively a HS is derived under the same assumption. Thus, a timed input sequence α is a *homing sequence* for a non-initialized complete observable possibly nondeterministic FSM with timed guards S if and only if the α/γ-successor of S for each output sequence γ is a singleton or the empty set. Proposition 1 and the one-to-one correspondence between states of an FSM with timed guards and its FSM abstraction imply the following statement.

Proposition 2. A timed input sequence α is a HS for a complete non-initialized observable possibly nondeterministic FSM with timed guards S if and only if the input sequence α_{FSM} is a HS for the FSM abstraction $\mathsf{A_S}$.

Therefore, a HS for an FSM with timed guards can be constructed as a HS for its FSM abstraction. At the next step, a HS for the FSM abstraction should be transformed into a timed HS for the FSM with timed guards.

Here we notice that if a complete deterministic FSM is reduced and strongly connected then a homing sequence always exists and length of a shortest homing sequence does not exceed $n(n – 1)/2$ where n is the number of FSM states. Since in this case, according to propositions below, the FSM abstraction of an FSM with timed guards possesses the same features, the same holds for TFSMs.

Proposition 3. A complete deterministic FSM with timed guards S is strongly connected if and only if the FSM abstraction $\mathsf{A_S}$ is a strongly connected FSM.

Proposition 4. A complete deterministic FSM with timed guards S is state reduced if and only if the FSM abstraction $\mathsf{A_S}$ is a reduced FSM.

Based on Propositions 2-4, the following statement holds.

Proposition 5. Given a complete deterministic reduced and strongly connected FSM with timed guards S, a homing sequence always exists for TFSM S and length of a shortest homing sequence does not exceed $n(n – 1)/2$ where n is the number of TFSM states.

Thus, a HS always exists for a complete reduced and strongly connected deterministic FSM with timed guards. Moreover, according to Proposition 4 and the results in [17], the upper bound of HS length cannot be reduced.

Note that for a non-initialized complete observable nondeterministic FSM length of a shortest homing sequence can reach $2^{n-1} – 1$ [16] and thus, the following statement holds due to Proposition 2.

Proposition 6. Given a non-initialized complete observable nondeterministic FSM with timed guards S, length of a shortest homing sequence can reach $2^{n-1} – 1$ where n is the number of TFSM states.

Similar to classical FSMs, a HS does not always exist for a nondeterministic observable FSM with timed guards. Checking the existence and deriving a HS for TFSMs can be performed by a slightly modified algorithm for FSMs [16].

Algorithm 1 for checking the existence and deriving a HS for an FSM with timed guards

Input: A complete non-initialized observable possibly nondeterministic FSM with timed guards $\mathsf{S} = (S, I, O, h_\mathsf{S})$

Output: Message 'There is no HS for S' or a HS α for TFSM S

Step 1. Derive the FSM abstraction $\mathsf{A_S} = (S, I_A, O, h_{\mathsf{AS}})$ for TFSM S.

Step 2. Construct a truncated successor tree for the FSM $\mathsf{A_S}$. The root of the tree is labeled by the set of all pairs of different states while the nodes of the successor tree are labeled by sets of pairs of different states from S or empty set; edges of the tree are labeled by inputs. There exists an edge labeled by an input i from a node labeled by the set P at level j, $j \geq 0$, to a node at level j+1 labeled by the set Q where a pair $\{s_1, s_2\} \in Q$ if and only if this pair is an *i*/*o*-successor of some pair of P. Given a node labeled by the set P at the level k, $k \geq 0$, the node is *terminal* if P is empty (**Rule 1**) or P contains a set R that labels a node at a level j, $j < k$ (**Rule-2**). If the successor tree has no nodes labeled with the empty set, then there is no HS for FSM A_S. An input sequence α_{FSM} that labels a path with minimal length to a node labeled with the empty set is transformed into a corresponding timed input sequence α that is a shortest HS for S.

Nevertheless, the number of successors for each node of a truncated successor tree when checking the HS existence equals the number of inputs of the FSM abstraction, i.e., reaches $(2(B_\mathsf{S} + 1)*|I|)$ instead of $|I|$ for classical FSMs. In the next section, we consider classes of TFSMs for which the number of the FSM abstraction inputs can be optimized when checking the existence and deriving a HS for a timed FSM.

4. Optimizing the number of the FSM abstraction inputs when checking the existence and deriving a HS for TFSMs

Here we notice that when constructing a HS for the FSM abstraction, the number of inputs becomes larger than that for the initial TFSM. In [11], approaches for the FSM abstraction optimization have been proposed in the context of test derivation procedures and in this section, we propose to use some of them for solving the HS problem for TFSMs.

Consider the FSM abstraction (Table 1) of the TFSM in Figure 1. As we can see it can well happen that there exist two equal rows for different inputs of the FSM abstraction. For example, the rows of the table for inputs $(i_1, [0, 0])$ and $(i_1, (0, 1))$ coincide and thus, it is sufficient to consider only one of them when looking for a HS.

Proposition 7. Given a complete FSM $\mathsf{S} = (S, I, O, h_\mathsf{S})$ and two inputs i and i', let for each state s it holds that $(s, i, o, s') \in h_\mathsf{S}$ implies $(s, i', o, s') \in h_\mathsf{S}$. The FSM S has a HS α if and only if a sequence α' obtained from α by the replacement of each input i' in α by i is a HS for the FSM $\mathsf{S}' = (S, I \setminus \{i'\}, O, h_{\mathsf{S}'})$ where $h_{\mathsf{S}'}$ is obtained from h_S by the removing transitions under input i'.

Proof. Let there exist HS $\alpha = i_1 \ldots i_l$ of FSM S. Therefore, for each trace $\alpha/\gamma = i_1/o_1 \ldots i_l/o_l$ the α/γ-successor of S is a singleton or does not exist. According to the statement conditions, for a sequence α' obtained from α by the replacement of each input i' in α by i it holds that the α'/γ-successor of S is a subset of the α/γ-successor of S, i.e., is a singleton or does not exist. Thus, α' is a HS for S'.

Corollary. The FSM S has a HS α of length l if and only if the FSM $\mathsf{S}' = (S, I \setminus \{i'\}, O, h_{\mathsf{S}'})$ a HS α' of length l.

In other words, if for a pair of inputs the rows of the transition table coincide then the latter can be deleted from the FSM without losing a homing sequence if such a sequence exists, i.e., it is a way to minimize the number of FSM inputs when solving the HS problem. For example, a transition table for the input reduced form of the FSM abstraction in Table 1 has three rows colored in grey and, respectively, the number of successors for each node of a successor tree is three (Figure 2). By direct inspection, one can assure that there is a HS (i_1, $(2, \infty)$), (i_1, $(2, \infty)$), (i_1, $(2, \infty)$) of length 3.

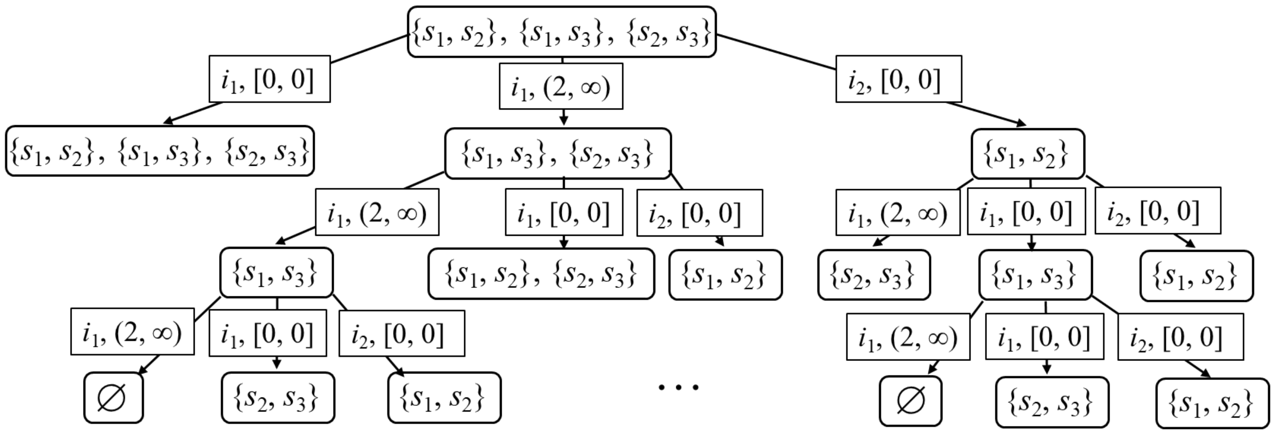

Fig. 2. The successor tree for the integer projection of FSM abstraction $\mathsf{A_S}$ (Table 1).

Let all the timed guards of a given TFSM S be closed on the left, i.e., each timed interval is of the form $[m, n)$. In this case, the *integer projection* $\mathsf{A}^{int}_{\mathsf{S}} = (S, I_A^{int}, O, h_{\mathsf{AS}})$ of the FSM abstraction where $I_A^{int} = \{(i, [m, m]) : i \in I, 0 \le m \le B_{\mathsf{S}}\}$ can be used for the HS derivation.

Proposition 8. Given a complete possibly nondeterministic TFSM $\mathsf{S} = (S, I, O, h_{\mathsf{S}})$ where each timed interval has the form $[m, n)$, let $\mathsf{A}^{int}_{\mathsf{S}} = (S, I_A^{int}, O, h^{int}_{\mathsf{AS}})$ where $I_A^{int} = \{(i, [m, m]) : i \in I, 0 \le m \le B_{\mathsf{S}}\}$ be the integer projection of the FSM abstraction. The FSM $\mathsf{A}^{int}_{\mathsf{S}}$ has a HS α_{FSM}' of length l if and only if TFSM S has a HS α of length l.

Proof. Let there exist a HS α of length l for FSM with timed guards S. By definition, α is a HS for TFSM S if and only if α_{FSM} is a HS for FSM abstraction $\mathsf{A_S}$. If all the timed guards of TFSM S are closed on the left, then for each input $(i, (a, b))$ of $\mathsf{A_S}$, there exists input $(i, [a, a])$ of $\mathsf{A}^{int}_{\mathsf{S}}$ such that $(s, (i, (a, b)), o, s') \in h_{\mathsf{AS}}$ if and only if $(s, (i, [a, a]), o, s') \in h^{int}_{\mathsf{AS}}$. Respectively, by Proposition 7, the FSM abstraction $\mathsf{A_S}$ has a HS α_{FSM} of length l if and only if FSM $\mathsf{A}^{int}_{\mathsf{S}}$ has a HS α'_{FSM} of length l.

Thus, the above proposition claims that when deriving a HS for a machine with timed guards, in some situations, the number of inputs of the FSM abstraction can be twice minimized without losing a solution.

5. Conclusions

In this work, a method for deriving a homing sequence for an FSM with timed guards is proposed based on its FSM abstraction. We show that similar to classical FSMs, length of a shortest HS is polynomial with respect to the number of states for a deterministic reduced FSM with timed guards and can reach an exponential value for the nondeterministic case. However, the FSM abstraction has more inputs than the initial TFSM and for this reason, we discuss how the number of inputs of the FSM abstraction can be optimized when solving the HS problem.

Acknowledgements

This work is partly supported by RFBR Project No. 19-07-00327.

References

1. Bochmann, G., Petrenko A.: Protocol testing: review of methods and relevance for software testing. In Proc. of International Symposium on Software Testing and Analysis, Seattle, 1994, pp. 109–123.
2. Lee, D., Yannakakis, M.: Testing finite state machines: state identification and verification. IEEE Trans. on Computers 43(3), 1994, pp. 306–320.
3. Gill, A.: Introduction to the Theory of Finite-State Machines. McGraw-Hill, 1962.
4. Jourdan, G.-V., Ural, H., and Yenigun, H.: Reduced checking sequences using unreliable reset. Inf. Process. Lett. 115(5), 2015, pp. 532–535.
5. H. Ural, F. Zhang, and J.C. Zhang.: Effects of overlapping subsequences in constructing checking sequences. Journal of Advances in Information Sciences 1(1), 2013, pp. 59–73.
6. Kushik N., López J., Cavalli A., Yevtushenko N.: Improving Protocol Passive Testing through "Gedanken" Experiments with Finite State Machines. In Proceedings of QRS, 2016, pp. 315–322.
7. Hung-En Wang, Kuan-Hua Tu, Jie-Hong R. Jiang, Natalia Kushik: Homing Sequence Derivation with Quantified Boolean Satisfiability. Lecture Notes in Computer Science (LNCS), № 10533, 2017, pp. 230–242.
8. Yenigun, H. Yevtushenko, N. Kushik, N. López J.: The effect of partiality and adaptivity on the complexity of FSM state identification problems. Trudy ISP RAN/Proc. ISP RAS 30 (1), 2018, pp. 7–24.
9. Petrenko, A., Yevtushenko N.: Adaptive Testing of Deterministic Implementations Specified by Nondeterministic FSMs. LNCS № 7019, 2011, pp. 162–178.
10. Krichen M. and Tripakis S. Conformance testing for real-time systems. Formal Methods Syst. Des. 34 (3), 2009, pp. 238–304.

11. El-Fakih K., Yevtushenko N., and Fouchal H.: Testing timed finite state machines with guaranteed fault coverage. Proc. of the 21st IFIP WG 6.1 Int. Conf. on Testing of Software and Communication Systems and 9th Int. FATES Workshop, 2009, pp. 66–80.
12. Merayo M.G., Nunez M., and Rodriguez I.: Formal testing from timed finite state machines. Comput. Networks: Int. J. Comput. Telecom. Networking 52 (2), 2008, pp. 432–460.
13. Bresolin D., El-Fakih K., Villa T., and Yevtushenko N.: Deterministic timed finite state machines: Equivalence checking and expressive power. Int. Conf. GANDALF, 2014, pp. 203–216.
14. Zhigulin M., Yevtushenko N., Maag S., Cavalli A.: FSM-Based Test Derivation Strategies for Systems with Time-Outs. Int. Conf. On Quality Software, Madrid, 2011, pp. 141–150.
15. Gromov, M., El-Fakih, K., Shabaldina, N., and Yevtushenko, N.: Distinguishing non-deterministic timed finite state machines. In Formal Techniques for Distributed Systems. Springer Berlin Heidelberg. Lecture Notes in Computer Science 5522, 2009, pp.137–151.
16. Kushik, N., Yevtushenko, N.: On the Length of Homing Sequences for Nondeterministic Finite State Machines. Lecture Notes in Computer Science, № 7982, 2013, pp. 220–231.
17. Hibbard T. N.: Lest upper bounds on minimal terminal state experiments of two classes of sequential machines. Journal of the ACM 8(4), 1961, pp. 601–612.

УДК 519.713.8

On some properties of timed finite state machines

Vinarskii E.M., Zakharov V.A. (Lomonosov Moscow State University)

Sequential reactive systems are formal models of programs that interact with the environment by receiving inputs and producing corresponding outputs. Such formal models are widely used in software engineering, computational linguistics, telecommunication, etc. In real life, the behavior of a reactive system depends not only on the flow of input data, but also on the time the input data arrive and the delays that occur when generating responses. To capture these aspects, a timed finite state machine (TFSM) is used as a formal model of a real-time sequential reactive system. However, in most of previously known works, this model was considered in simplified semantics: transduction relations of TFSMs are defined in such a way that the responses in the output stream, regardless of their timestamps, follow in the same order in which the corresponding inputs are delivered to the machine. This simplification makes the model easier to analyze and manipulate, but it misses many important aspects of real-time computation. In this paper we study a more realistic semantics of TFSMs and show how to represent it by means of Labeled Transition Systems. This opens up a possibility to apply traditional formal methods for verifying more subtle properties of real-time reactive behavior which were previously ignored.

Keywords: *Timed finite state machines, transduction relation, safety property, Labeled Transition System, bisimulation*

1. Introduction

Timed finite state machines (TFSMs) are, perhaps, the most simple extensions of finite state machines (FSMs) which are widely used for modelling and analysis of real-time reactive systems. There are several known ways to generalize the concept of finite state machines for modeling real time computations. One of the most advanced is the concept of Timed Automata (TA) [1]. In [2–4] it was shown that TAs supplied with clocks (timers), timed guards at their transitions and timed invariants at their states capture many important aspects of real-time computations. However, the behavior of TAs is difficult for the analysis since TAs were given too much freedom in handling their timers. Complex manipulations with timers are unavoidable when it is necessary to synchronize several events at once (e.g. to output the next control signal no more than 2 time units after receiving the input data, but no earlier than 1 time unit after the previous control signal is issued).

However, in many practically important cases, the behavior of control systems is not so complicated, and the response of a system is determined exclusively by the current control state and the parameters of the input data. Therefore, in order to avoid the difficulties arising when working with such a complex computational model as TAs, the authors of [7, 10, 11] extended a FSM with a single timer and distinguished three families of TFSMs depending on admissible *modus operandi* with timers: (i) a TFSM with timed guards which fire transitions only when a timestamp of an input satisfies the corresponding time guard; (ii) a TFSM with timeouts which force the machine to make a transition when a prescribed waiting time expires; (iii) a TFSM with timed guards and timeouts. Whenever a TFSM moves to a new state, it resets its timer. It was shown that TFSMs of this kind may be used as adequate formal models for many interesting applications where real-time effects are concerned, and they are also admit efficient algorithms for analysis and manipulations.

As a model of real time reactive system a TFSM converts timestamped input streams into timestamped output streams. A distinguishing feature of those classes of TFSMs that have been studied in [7, 10, 11] is that the order of the outputs is determined not by their timestamps, but by the order of the corresponding inputs. This principle alleviates substantially verification and optimization of TFSMs, but it also makes such a model inadequate for many applications. Consider, for example, a behavior of a controller in Software Defined Networks (SDN) [9]. It supplies flow tables of network switches with packet forwarding rules. Packets arrived to an input buffer of a switch are matched against a flow table to select an appropriate rule which either forwards the packet to some output buffer, or drops it. When a controller updates flow tables, the order in which new rules are set in a table may differ from the order in which they were sent by the controller due to delivery delays. When the controller, say, receives requests R_1, R_2 from the network it responds with a pair of commands A_1, A_2 which add new entries to a flow table. It may happen so (see [13]) that the controller at receiving a sequence of timestamped inputs (input timed word) $(R_1, t_1), (R_2, t_2)$, where $t_1 < t_2$, computes an output timed word $(A_2, \tau_1), (A_1, \tau_2)$, where $\tau_2 < \tau_1$, i.e. the second rule will be set before the first one. This incorrectness cannot be detected using the early variants of TFSMs.

To cope with this shortcoming of conventional TFSMs and to make their behaviour more “realistic” some improvements to TFSMs’ semantics were introduced in [12]. A new model of TFSMs captures an important feature of reactive system computations: the order of responses in the output stream is determined by the time they were generated, not the order of the corresponding requests in the input stream: if a request b arrives at the input of a machine after a signal a then it is possible that a response to a follows a response to b. An improved semantics of TFSMs allows one to represent explicitly the erroneous behaviour of SDN controller mentioned above. But the new semantics of TFSMs loses the incremental property: when an input α' extends an input α it is not necessary that the corresponding output β' extends an output β which results from α. The lack of this fine property significantly complicates the solution of the verification problem if we use only the transition relations in TFSM programs.

To overcome this fundamental difficulty we introduce an alternative semantics of TFSMs that better accommodate traditional verification techniques. This semantics is defined in terms of Labeled Transition Systems (LTSs) based on the concept of TFSM configuration. However, the statespace of such $LTS(\mathcal{T})$ corresponding to a TFSM $\mathcal{T}$ is uncountable. Our long term goal is to develop an algorithm which for every TFSM $\mathcal{T}$ constructs a finite $LTS_{fin}(\mathcal{T})$ which simulates $LTS(\mathcal{T})$ and, thus, can be used for verification of $\mathcal{T}$. In this paper we present some preliminary results of our research.

In Section 2 and 3 we introduce the concept of TFSMs with the improved automata-theoretic semantics, and demonstrate how it can be used to adequately describe the behavior of SDN controllers. In Section 4 we define LTS-based semantics for TFSMs under consideration, study its relationship with an automata-theoretic semantics, and present some results that contribute to reducing the size of $LTS(\mathcal{T})$ and building $LTS_{fin}(\mathcal{T})$. Section 5 concludes the paper and presents some avenues for future work.

This research is supported by RFBR project No. 18-01-00854.

2. Preliminaries

Consider two non-empty finite alphabets A and B; the alphabet A is called an *input* alphabet while B is an *output* alphabet. The symbols of A can be viewed as control signals (inputs) received by some real-time system, and the symbols of B may be regarded as responses (outputs) generated by the system. A finite sequence $\alpha = a_1, a_2, \dots, a_n$ of inputs is called an *input word*, whereas a sequence $\beta = b_1, b_2, \dots, b_n$ of outputs is called an *output word*.

For modeling time we use a special variable which takes values in the non-negative real domain $\mathbb{R}_0^+$. Temporal aspects of computations are defined with the help of such notions as timestamps, time sequences, time guards and delays. A *timestamp* is a positive real number from $\mathbb{R}^+$ which indicates a time instance when some event occurs. A *time sequence* is any increasing sequence of timestamps. A *time guard* (and a *delay*) is an interval $g = \langle u, v \rangle$, where $\langle \in \{(, [\}$, $\rangle \in \{),]\}$, and u, v are such numbers from $\mathbb{R}^+$ that $u \leq v$. Time guards and delays are used to specify periods of time in which some events are possible. We call u the *left bound* of a timed guard (delay) g while v is the *right bound* of this timed guard (delay). If a delay is a singleton $[d, d]$ then we say that it is a *sharp delay* (and write d instead of $[d, d]$)

Let $w = x_1, x_2, \dots x_n$ and $\tau = t_1, t_2, \dots, t_n$ be an input (output) word and an increasing time sequence of the same length. Then any pair (x_i, t_i) is called an input (output) *timed symbol*, and a pair $\alpha = (w, \tau)$ is called a *timed word*. We denote by $t(\alpha)$ the last value τ_n in the sequence τ. Let A and B be an input and an output alphabets. Denote by G and D some sets of time guards and delays. Then a *Timed Finite State Machine* (TFSM) over A, B, G, and D is a triple $\mathcal{T} = (S, \rho, s_0)$ where S is a finite non-empty *set of states*, $\rho \subseteq (S \times A \times G \times B \times D \times S)$ is a finite *transduction relation*, s_0 is an *initial state*.

Every 6-tuple (s, a, g, b, d, s') in ρ is called a *transduction* of $\mathcal{T}$. Such a transduction is an atomic action that a TFSM $\mathcal{T}$ can perform: if after time t since the machine is into a state s, a signal a arrives at its input and the guard condition $t \in g$ for triggering the transduction is satisfied, then $\mathcal{T}$ immediately passes to the state s' and the output response b is produced after time $t' = t + \delta$ where $\delta \in d$. A TFSM applies these actions to all timed inputs (a_i, t_i) of a given input timed word (w, τ) and generates, as a result, an output timed word (z, τ'). This behavior of TFSMs is defined formally as follows.

A *run* of a TFSM $\mathcal{T}$ on an input timed word $\alpha = (a_1, t_1), (a_2, t_2), \dots, (a_n, t_n)$ is a finite sequence $r = s_0 \xrightarrow{(a_1,t_1)/(b_1,\tau_1)} s_1 \xrightarrow{(a_2,t_2)/(b_2,\tau_2)} \dots \xrightarrow{(a_n,t_n)/(b_n,\tau_n)} s_n$ of 6-tuples from $S \times A \times \mathbb{R}^+ \times B \times \mathbb{R}^+ \times S$ such that for each $i \in \{1, \dots, n\}$ there exists a transduction $(s_{i-1}, a_i, g_i, b_i, d_i, s_i)$ of $\mathcal{T}$ which complies with the following requirements: 1) $t_i - t_{i-1} \in g_i$, and 2) $\tau_i = t_i + \delta$, where $\delta \in d_i$. In this case we say that a run r of a TFSM $\mathcal{T}$ *converts* the input timed word α to the output timed word $\beta = (b_{j_1}, \tau_{j_1}), (b_{j_2}, \tau_{j_2}), \dots, (b_{j_n}, \tau_{j_n})$ which is the permutation of the sequence $(b_1, \tau_1), (b_2, \tau_2), \dots, (b_n, \tau_n)$. For every TFSM $\mathcal{T}$ we denote by $TR(\mathcal{T})$ a *transduction relation* computed by $\mathcal{T}$ which is the set of all pairs (α, β), where α is an input timed word, and β is the result of conversion of α by $\mathcal{T}$. If all delays in the set D are sharp then we say that $\mathcal{T}$ is a *TFSM with sharp delays*. In this paper we consider only TFSMs with sharp delays.

Consider, for example, a TFSM $\mathcal{T}$ shown on Fig. 1 and a run r induced by the input timed word $\alpha = (a_1, 1.5), (a_2, 2.7), (a_3, 4)$. Then this run converts α to the output timed word $\beta = (b_2, 3.7), (b_1, 4.5), (b_3, 6)$.

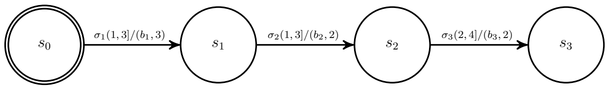

Figure 1. A TFSM $\mathcal{T}$

3. Safety property for real-time systems

In this section, we show that for the modified semantics of TFSMs as defined above, the analysis of safety properties turns out to be difficult. A subset $P_{safe} \subseteq A^\omega$ is called a *safety property* (see [5]) if every ω-word $\alpha \in A^\omega \setminus P_{safe}$ has such a finite prefix β that $\beta\alpha' \notin P_{safe}$ holds for any ω-word α', i.e. if a run α of a system is unsafe then a safety violation can be detected after a finite number of steps β, and no further system actions α' can eliminate this error. A straightforward extension of this concept to the case of real-time computations brings us to the following definition. A subset $P_{safe} \subseteq (A \times \mathcal{R}^+)^\omega$ is called a *real-time safety property* if for every timed ω-word $\alpha \in (A \times \mathcal{R}^+)^\omega \backslash P_{safe}$ there exists such a finite timed prefix β of α that $\alpha' \notin P_{safe}$ holds for any timed extension α' of β.

This is a reasonable and intuitive definition, but when it comes to TFSMs operating within the improved semantics, it brings also some unpleasant effects that hinder verification of these real time models of computation. Usually *safety* means that no errors happen in the course of a run of a model, and if such an error is detected after some finite number of steps, then it can be announced regardless of the subsequent steps of the computation. However, this common and useful feature of safety checking does not always hold in the case of TFSMs. We illustrate this phenomenon with an example. Consider an SDN controller $\mathcal{C}$ which receives requests to update a flow table of an SDN switch and generates in response the appropriate instructions. Since the formation and delivery of an instruction to a network switch takes some time, the responses start to take effect with some delay. A TFSM which is a real-time model of such a controller is depicted on Fig. 2. It operates with the set of inputs (requests) PF_i (put a rule r_i into a flow table), GF_i (get an info about a rule r_i from a flow table), DF_i (delete a rule r_i from a flow table), and the set of outputs (responses) FA_i (a rule r_i is inserted into a flow table), FR_i (info about a rule r_i is received from a flow table), FD_i (a rule r_i is erased in a flow table).

The safety property that the designers of a controller wants to respect requires that flow table update instructions must be always activated the same order as flow table update requests are received. When TFSM $\mathcal{C}$ receives an input timed word $\alpha = (GF_1, 1.5), (PF_1, 3.7), (PF_2, 5.5)$ it converts it to an output timed word $\beta = (FR_1, 3.5), (FA_2, 8.5), (FA_1, 10.7)$. Clearly, such a

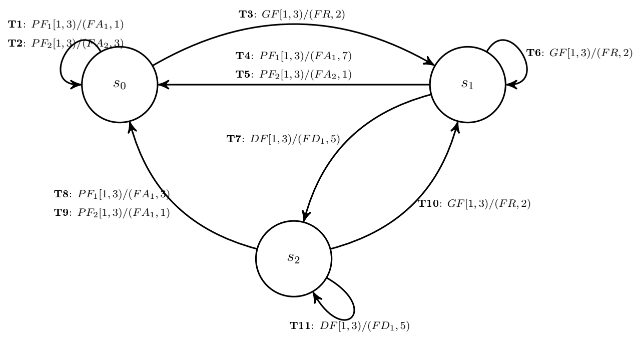
$\mathbf{T1}$: $PF_1[1,3)/(FA_1,1)$
$\mathbf{T2}$: $PF_2[1,3)/(FA_2,3)$
$\mathbf{T3}$: $GF[1,3)/(FR,2)$
$\mathbf{T4}$: $PF_1[1,3)/(FA_1,7)$
$\mathbf{T5}$: $PF_2[1,3)/(FA_2,1)$
$\mathbf{T6}$: $GF[1,3)/(FR,2)$
s_0
s_1
$\mathbf{T7}$: $DF[1,3)/(FD_1,5)$
$\mathbf{T8}$: $PF_1[1,3)/(FA_1,3)$
$\mathbf{T9}$: $PF_2[1,3)/(FA_1,1)$
$\mathbf{T10}$: $GF[1,3)/(FR,2)$
s_2
$\mathbf{T11}$: $DF[1,3)/(FD_1,5)$

Figure 2. An SDN Controller

behavior does not satisfies the safety requirement. But if $\mathcal{C}$ continues to receive input signals $\alpha' = (GF_1, 1.5), (PF_1, 3.7), (PF_2, 5.5), (PF_1, 6.7)$ then the corresponding output timed word $\beta' = (FR_1, 3.5), (FA_1, 7.7), (FA_2, 8.5), (FA_1, 10.7)$ will indicate that the satisfiability of the safety property has been restored. To cope with such cases, TFSMs need to be provided with a more suitable way for presenting their behavior.

4. From TFSM configurations to a Labeled Transition System

Operational semantics of TFSMs defined so far suits well the modeling of real-time reactive information processing systems since it quite naturally captures many important effects of real-time computations. However, this semantics is unfavorable for the application of traditional methods and tools for analyzing the behavior of real-time systems, since it lacks the concept of a state of computation as a snapshot of a computing process. In this section to overcome this drawback we introduce a concept of a configuration which makes it possible to represent TFSMs' behaviors by means of Labeled Transition Systems on such configurations.

4.1. Configurations of Timed Finite State Machine

Intuitively, a configuration of a TFSM $\mathcal{T}$ is a snapshot of a TFSM's computation which includes 3 components: a control state of a TFSM $\mathcal{T}$, time elapsed since the last input (timer value), and a set of output timed symbols that have been generated but not yet issued due to delays. Formally, a configuration q is a triple $\langle s, t, TC \rangle$ such that:

- $s \in S$ is a state of $\mathcal{T}$; it will be referred as $q.s$;
- $t \in \mathcal{R}_0^+$ is time elapsed since the last input (referred as $q.t$);
- $TC = \{(b_1, \tau_1), (b_2, \tau_2), \dots, (b_n, \tau_n)\}$, where $(b_i, \tau_i), 1 \leq i \leq n$ are timed output symbols, is an *output timed context* of q (referred as $q.c$).

A *capacity of a configuration* q is the cardinality $|q.c|$ of its output timed context. We denote by $q.T$ a value $min(\tau_1, \ldots, \tau_n)$ if $q.c \neq \emptyset$; in the case of $q.c = \emptyset$ we assume that $q.T = \infty$. We denote by $Q(\mathcal{T})$ the set of all possible configurations of a TFSM $\mathcal{T}$.

4.2. Labeled Transition Systems

For any given TFSM $\mathcal{T}$ over an input alphabet A and an output alphabet B we define a Labeled Transition System $\mathcal{LTS}(\mathcal{T})$ which has three types of transitions on configurations: (i) time advancement, (ii) input read, and (iii) output write.

Formally, a *Labeed Transition System* $\mathcal{LTS}(\mathcal{T})$ of a TFSM $\mathcal{T} = (S, \rho, s_0)$ over alphabets A and B is a triple $(Q(\mathcal{T}), q_0, \rho_{\mathcal{LTS}})$ where $q_0 = \langle s_0, 0, \emptyset \rangle$ is the *initial configuration*, and $\rho_{\mathcal{LTS}}$ is a transition relation of the type $(Q \times \mathcal{R}^+ \times Q) \cup (Q \times A \times Q) \cup (Q \times B \times Q)$ which is defined for every configuration $q = \langle s, t, \{(b_1, \tau_1), (b_2, \tau_2), \ldots, (b_m, \tau_m)\} \rangle$ as follows.

1. For every $\delta \in \mathcal{R}^+$ such that $\delta \leq q.T$ there exists a time advancement transition $q \xrightarrow{\delta} q'$ in $\rho_{\mathcal{LTS}}$, where $q' = \langle s, t + \delta, \{(b_1, \tau_1 - \delta), (b_2, \tau_2 - \delta), \ldots, (b_m, \tau_m - \delta)\} \rangle$.
2. For every transduction $(s, a, \langle u, v \rangle, c, d, s')$ of $\mathcal{T}$ such that $t \in \langle u, v \rangle$ there exists an input transition $q \xrightarrow{a} q'$ in $\rho_{\mathcal{LTS}}$, where $q' = \langle s', 0, q.c \cup \{(c, d)\} \rangle$; in this case, we say that this input transition is *induced* by this transduction.
3. For every pair $(b, 0) \in q.c$ there exists an output transition $q \xrightarrow{b} q'$ in $\rho_{\mathcal{LTS}}$, where $q' = \langle s, t, q.c \setminus \{(b, 0)\} \rangle$.

A *path* in $\mathcal{LTS}(\mathcal{T})$ is any sequence of transitions $\pi = q_0 \xrightarrow{x_1} q_1 \xrightarrow{x_1} \cdots q_{k-1} \xrightarrow{x_k} q_k$. A path π is called *complete* if $q_k.c = \emptyset$. With every path π in $\mathcal{LTS}(\mathcal{T})$ we associate a pair $TR(\pi) = (\alpha, \beta)$ of an input and an output timed words which is defined by induction on the length of π according to the following rules.

1. If π is an empty path then $TR(\pi) = (\varepsilon, \varepsilon)$;
2. Suppose that π' is a path in $\mathcal{LTS}(\mathcal{T})$ from q_0 to q' such that $TR(\pi') = (\alpha', \beta')$, and a path π is an extension of π' with a transition $E = q' \xrightarrow{x} q$. If E is
   - a time advancement transition then $TR(\pi) = TR(\pi')$;
   - an input transition then $TR(\pi) = (\alpha, \beta')$, where $\alpha = \alpha', (x, t(\alpha') + q'.t)$;
   - an output transition then $TR(\pi) = (\alpha', \beta)$, where $\beta = \beta', (x, t(\alpha') + q'.t)$.

A transduction relation $TR(\mathcal{LTS}(\mathcal{T}))$ specified by a $\mathcal{LTS}(\mathcal{T})$ is the set of all pairs $TR(\pi)$ associated with all complete paths π in $\mathcal{LTS}(\mathcal{T})$. As it may be seen, $\mathcal{LTS}(\mathcal{T})$ thus defined completely characterizes the behavior of a TFSM $\mathcal{T}$.

Proposition 1. $TR(\mathcal{T}) = TR(\mathcal{LTS}(\mathcal{T}))$ holds for every TFSM $\mathcal{T}$.

Proof (draft). Clearly, every finite path in $\mathcal{LTS}(\mathcal{T})$ can be extended to a complete path; so, $\mathcal{LTS}(\mathcal{T})$ has no useless paths. The inclusion $TR(\mathcal{T}) \subseteq TR(\mathcal{LTS}(\mathcal{T}))$ is proved straightforwardly by constructing an appropriate complete path in $\mathcal{LTS}(\mathcal{T})$ for every finite run of $\mathcal{T}$. The inclusion $TR(\mathcal{LTS}(\mathcal{T})) \subseteq TR(\mathcal{T})$ can be proved by induction on the length of an input timed word α.

The main advantage of $\mathcal{LTS}(\mathcal{T})$ is that this a model of the same type as that used for verification of timed automata [1]. This makes it possible to apply such model checking tools as Uppaal [6] for the analysis of behavior of TFSMs. However, such an analysis is fraught with certain difficulties: the statespace of $\mathcal{LTS}(\mathcal{T})$ may be infinite. Partially, this is due to the fact that the capacity of configurations in $\mathcal{LTS}(\mathcal{T})$ is unbounded in general case. Nevertheless, under certain conditions some bounds on the capacity of configurations of TFSMs can be established.

Let u, v be a pair of real numbers such that $0 < u \leq v$. A TFSM $\mathcal{T} = (S, \rho, s_0)$ is called (u,v)*-progressive* if for every transduction $(s, a, g, b, d, s') \in \rho$ a guard $g = (u', v')$ is such that $u < u'$ and $v' < v$, i.e. the guards of all transductions of $\mathcal{T}$ are within the interval (u, v).

Proposition 2. If $\mathcal{T}$ is a (u,v)-progressive TFSM with sharp delays then there exists such an integer ℓ that $|q.c| < \ell$ holds for all configurations $q \in Q(\mathcal{T})$ reachable in $\mathcal{LTS}(\mathcal{T})$ from the initial configuration q_0.

Proof (draft). Denote by d_{max} the maximal sharp delay in the transductions of $\mathcal{T}$, and let $\ell = \lceil \frac{d_{max}}{u} \rceil$. Then by *redictio ad absurdum* it can be shown that for every path $\pi = q_0 \xrightarrow{x_1} q_1 \xrightarrow{x_1} \cdots q_{k-1} \xrightarrow{x_k} q_k$ in $\mathcal{LTS}(\mathcal{T})$ the inequality $|q_i.c| \leq \ell$ holds for every $1 \leq i \leq k$.

Another difficulty in analyzing the models $\mathcal{LTS}(\mathcal{T})$ of TFSMs is a kind of livelock effect (see [5]) when any extension of a path in $\mathcal{LTS}(\mathcal{T})$ is achieved by time advancement transitions only. To detect and exclude from further consideration such redundant paths, we will use the notion of timelock configuration (see [5]). For every state s of a TFSM $\mathcal{T}$ denote by $up(s)$ the maximal upper bound v in the guards $g = (u, v)$ of all transductions (s, a, g, b, d, s') that are enable in s. A configuration q is called a *timelock configuration* if $q.c = \emptyset$ and $q.t > up(q.s)$. All other configurations in $Q(\mathcal{T})$ are called *progressive*. As it may be seen from this definition, only progressive configurations matter: if a path in $\mathcal{LTS}(\mathcal{T})$ reaches a timelock configuration then there are no outputs pending and no inputs will be able to trigger any further transduction of $\mathcal{LTS}(\mathcal{T})$. Therefore, no further analysis of paths outgoing from timelock configuration is necessary. Detection of timelock configurations is easy for (u, v)-progressive TFSMs.

Proposition 3. For every state s of a (u, v)-progressive TFSM $\mathcal{T}$ there exists $c_s \geq 0$ such that every configuration $q = (s, t, TC)$ is a timelock configuration iff $q.t > c_s$

Proof. Let s be a state of $\mathcal{T}$ and let $v = ub(s)$. Consider a set of configurations $Q_s = \{q \in Q \mid (q.s = s) \ \wedge (q.t \geq v) \ \wedge (q.c = \emptyset)\}$ and the corresponding set of timestamps $TS_s = \{t| \ (q \in Q_s) \ \wedge (q.t = t)\}$. It easy to see that TS_s has the infimum c_s which satisfies the assertion of the proposition.

Our next step will be to find out or introduce effectively a finitely indexed equivalence relation $\sim$ on the set of progressive configurations of $\mathcal{LTS}(\mathcal{T})$ which makes it possible to construct such a finite model $\mathcal{LTS}_{fin}(\mathcal{T}) = \mathcal{LTS}(\mathcal{T})/\sim$ that $\mathcal{LTS}_{fin}(\mathcal{T})$ simulates $LTS(\mathcal{T})$ w.r.t. some suitable simulation relation. Such a finite model $\mathcal{LTS}_{fin}(\mathcal{T})$ opens a way for applying conventional model checking tools for verification of reactive systems represented by TFSMs of a new type.

5. Conclusion and future work

In this paper we showed that timed finite state machines which generate responses to the input signals in the order that corresponds to the output delays is a quite adequate model of real-time reactive computing systems and it could be used in some applications. However, it turned out that some safety properties that arise in such applications are difficult to analyze based on the conventional operational semantics for TFSMs. To overcome this trouble we adapt the operational semantics of TFSMs to the concept of Labeled Transition Systems to take advantage of well-known verification techniques for models of real-time systems.

Since $LTS(\mathcal{T})$ which represents all possible runs of a TFSM $\mathcal{T}$ may have infinitely many internal states (configurations), our next step we are going to made in the future work is to develop a technique for converting infinite $LTS(\mathcal{T})$ to finite $\mathcal{LTS}_{fin}(\mathcal{T})$ in such a way that $\mathcal{LTS}_{fin}(\mathcal{T})$ simulates $LTS(\mathcal{T})$ w.r.t. some suitable simulation relation. We expect to find out an appropriate simulation relation which (i) preserves the most important properties of TFSM behaviours, and (ii) admits an efficient translation of a TFSM $\mathcal{T}$ into its finite model $\mathcal{LTS}_{fin}(\mathcal{T})$. We think that such a translation could be developed with the help of configuration patterns – a new concept which would play the same role for model checking of TFSMs as timed regions for the analysis of timed automata (see [1]).

The authors of the article are grateful to the anonymous reviewers for useful comments that helped to improve the article.

References

1. Alur R., Dill D. A theory of timed automata // Theoretical Computer Science. 1994. Vol. 126, № 2. P. 183–235.
2. Alur R., Madhusudan P. Decision Problems for Timed Automata // Proceedings of the 4-th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM'04). 2004. P. 1–24.
3. Asarin E., Caspi P., Oded M. Timed Regular Expressions // Journal of the ACM. 2001. Vol. 49, № 2. P. 1–35.
4. Asarin E., Caspi P., Oded M. A Kleene theorem for timed automata // Proceedings of the 12-th Annual IEEE Symposium on Logic in Computer Science (LICS'97). 1997. P. 160–171.
5. Baier C., Katoen J. Principles of Model Checking. Cambridge: MIT Press Cambridge. 2008. 994 p.
6. Behrmann G., David A., Larsen K.G. A tutorial on Uppaal // Proceedings of the International School on Formal Methods for the Design of Computer, Communication, and Software Systems. 2004. P. 200–236.
7. Bresolin D, El-Fakih K., Villa T., Yevtushenko N. Deterministic Timed Finite State Machines: Equivalence Checking and Expressive Power // Proceedings of the 5-th International Symposium on Games, Automata, Logics and Formal Verification. 2014. P. 203–216.
8. Gill A. Introduction to the Theory of Finite-state Machines. New York: McGraw-Hill, 1962, 207 p.
9. McKeown N., Anderson T., Balakrishnan H., et. al. OpenFlow: enabling innovation in campus networks // ACM SIGCOMM Computer Communication Review, 2008. Vol. 38, № 2. P. 69–74.
10. Tvardovskii A., Yevtushenko N. Minimizing finite state machines with time guards and timeouts // Proceedings of ISP RAS. 2017. V. 29, № 4. P. 139–154
11. Tvardovskii A., Yevtushenko N. Minimizing Timed Finite State Machines // Tomsk State University Journal of Control and Computer Science. 2014. Vol. 29, № 4. P. 77–83.
12. Vinarskii E., Zakharov V. On the verification of strictly deterministic behaviour of Timed Finite State Machines// Proceedings of ISP RAS. 2018. Vol. 30, № 3. P. 325–340.
13. Vinarskii E., López J., Kushik N., Yevtushenko N., Zeghlache M. A Model Checking Based Approach for Detecting SDN Races // Proceedings of the 31-st IFIP WG 6.1 International Conference on Testing Software and Systems – ICTSS. 2019. P. 194–211.

УДК 519.713.8

Using an extension of CTL^* for specification and verification of sequential reactive systems

Gnatenko A.R., Zakharov V.A.

(National Research University Higher School of Economics)

Sequential reactive systems such as controllers, device drivers, computer interpreters operate with two data streams and transform input streams of data (control signals, instructions) into output streams of control signals (instructions, data). Finite state transducers are widely used as an adequate formal model for information processing systems of this kind. Since runs of transducers develop over time, temporal logics, obviously, could be used as both simple and expressive formalism for specifying the behavior of sequential reactive systems. However, the conventional applied temporal logics (HML, LTL, CTL, μ-calculus) do not suit this purpose well, since their formulae are interpreted over ω-languages, whereas the behavior of transducers are represented by binary relations on infinite sequences, i.e. by ω-transductions. To provide temporal logics with the ability to specify the property of transductions that characterize the behavior of reactive systems, we introduced new extensions of these logics. Two principal features distinguish these extension: 1) temporal operators are parameterized by sets of streams (languages) admissible for input, and 2) sets (languages) of expected output streams are used as basic predicates. In our previous papers [7, 8, 13] we studied the expressive power and the model checking problem for *Reg*-LTL and *Reg*-CTL which are the extensions of LTL and CTL where the languages mentioned above are regular ones. We discovered that parametrization of this kind increases expressive power of temporal logics though retains the decidability of the model checking problem. Our next step in the systematic exploration of new extensions of temporal logics intended for specification and verification of sequential reactive systems is the study of the model checking problem for finite state transducers against *Reg*-CTL^* formulae. In this paper we develop a model checking algorithm for *Reg*-CTL^* and show that this problem is in ExpSpace.

Keywords: *reactive system, model checking, finite state transducer, temporal logic, regular language, specification, verification*

1. Introduction

Finite state machines are widely used in computer science as models of sequential computing systems. In particular, finite state transducers serve as a suitable formal model for various software and hardware systems such as controllers, device drivers, network switches, computer interpreters, etc. which operate with two data streams. These devices and programs receive streams of data (control signals, instructions) at their inputs and transform them into output streams of control signals (instructions, data). Hardware devices of this type include adapters, network switches, controllers. In software engineering transducers are used as formal models of various programs and protocols that manipulate with strings of symbols, flows commands, data streams, etc. (see [2, 11, 19]). Such programs, systems and devices can be grouped under the general name *sequential reactive systems*. A sequential reactive system operates in discrete time. At each step of computation it receives a control signal (or a piece of input data) from the environment and generates (performs, outputs) a sequence of actions (or a piece of output data) in response. Those output actions and the order of their performance depend on the received input signal, but also on all the previous control signals.

We focus on so called *online systems* which are supplied with only finite memory. In [7] we presented and discussed a number of examples to show that finite state transducers is a simple albeit rather adequate formalism for modeling the behavior of reactive sequential systems in many applications. The behavior of such systems is characterized not by a set of sequences of events, but by a relationship between two sequences of events. A typical property of such a behavior that needs verification is that for for each input word of a given pattern the transducer always outputs a word of another given pattern. The requirements of this kind can be formally specified by means of temporal logics adapted for reasoning about pairs of sequences of events.

Such temporal logics were introduced and studied in [6, 7, 13]. In [13] a new extension of Linear Temporal Logics (LTL) was introduced as a formal language for specification of the behavior of sequential reactive systems. In this logic $\mathcal{LP}$-LTL the temporal operators are parameterized by sets of words (languages) that represent distinguished flows of control signals that impact on a reactive system. Basic predicates in $\mathcal{LP}$-LTL are also languages in the alphabet of basic actions of a transducer; they represent the expected response of a transducer to the specified environmental influences. In [13] the authors studied the model checking problem for regular fragment Reg-LTL of this logic when only regular languages are used as basic predicates and parameters of temporal operators. It was shown that the model checking problem for finite state transducers against the formulae of Reg-LTL is decidable in double exponential time. In [7] we estimate the expressive power $\mathcal{LP}$-LTL by comparing it with some well known logics widely used in computer science for specification of reactive systems behavior. In [6] a model checking algorithm was proposed for *Reg-CTL* which is a regular extension of Computational Tree Logic *CTL*.

In this paper we consider a more general specification language *Reg-CTL** which is a regular extension of Generalized Computation Tree Logic *CTL**, develop a model checking algorithm for this logic and estimate its complexity. This is the main contribution of the paper. The results obtained are based on the methods developed in [6, 8, 13]; they are to continue the line of research initiated in these papers. In Section 2 we introduce the formal definition of a finite state transducers as well as the syntax and the semantics *Reg-CTL**. In Section 3 we give a short survey of some previously known extensions of conventional temporal logics and compare them with our family of logics. In Section 4 we propose a model checking procedure for *Reg-CTL** and estimate its complexity. Section 5 is left for the comparative analysis of the results obtained and similar decisions of model checking problem for other extensions of temporal logics, and also for a discussion on the further lines of research.

This research is supported by RFBR project No. 18-01-00854.

2. Reactive systems models and their specifications

Sequential reactive systems such as adapters, controllers, device drivers, computer interpreters operate with two data streams and transform input streams of data (control signals, instructions, etc.) into output streams of data (control signals, instructions, etc.). Such mappings are called *transduction relations*, and finite state transducers are widely used as an adequate formal model for information processing systems of this kind.

Let $\mathcal{C}$ and $\mathcal{A}$ be finite alphabets. The elements of $\mathcal{C}$ are called *input signals* and the elements of $\mathcal{A}$ are *basic actions*. Denote by $\mathcal{A}^*$ the set of all finite words over $\mathcal{A}$, which are called *compound actions*. Given two compound actions u and v, we write uv for their concatenation, and ε for the empty word.

A *finite state transducer* over the $\mathcal{C}$ and $\mathcal{A}$ is a quintuple $\pi = (Q, \mathcal{C}, \mathcal{A}, q_{init}, T)$, where Q is a finite set of *control states*, $q_{init} \in Q$ is the *initial state*, and $T \subseteq Q \times \mathcal{C} \times Q \times \mathcal{A}^*$ is a total *transition relation*. The *size* $|\pi|$ of a transducer π is the size of a table description of its transition relation T. A *transition* $\tau = (q', c, q'', h) \in T$ means that a transducer is capable to output a compound action h and pass control to a state q'' at receiving an input signal c in a state q'. A *trajectory* of π from a state q_0 is any infinite sequence of transitions $tr = \{\tau_i\}_{i \geqslant 1}$ such that $\tau_i = (q_{i-1}, c_i, q_i, h_i) \in T$ for all i, $1 \leqslant i \leqslant n$. A sequence of pairs $(c_1, h_1), (c_2, h_2), \ldots$

of a trajectory tr displays a behavior of a transducer as seen by an outside observer. We denote by $Tr_\pi(q)$ the set of all trajectories of π from a state q, and write simply Tr_π in the case when $q = q_{init}$. We denote by $\text{Fin}(Tr_\pi(q))$ the set of all finite prefixes of trajectories in $Tr_\pi(q)$; such prefixes will be called *finite trajectories* from a state q.

To be able to analyze the behavior of a transducer π, it is advisable to use a structure that represents all runs of π; it is obtained as an unfolding of the transition relation of π. A *computation graph* of a finite state transducer $\pi = (Q, \mathcal{C}, \mathcal{A}, q_{init}, T)$ is a labeled digraph $\Gamma_\pi = (V, E, v_{init})$, which has the set of nodes $V = Q \times \mathcal{A}^*$ and the set of labeled arcs $E \subseteq V \times \mathcal{C} \times V$, such that for every pair of nodes $u = (q', s')$ and $v = (q'', s'')$ the following relationship holds

$$(u, c, v) \in E \quad \Longleftrightarrow \quad \exists h \in \mathcal{A}^* \text{ such that } (q', c, q'', h) \in T \text{ and } s'' = s'h.$$

The node $v_{init} = (q_{init}, \varepsilon)$ is called the *initial node* of Γ_π. As it can be seen from the relationship above, the correspondence between trajectories of π and paths in Γ_π is as follows: for every state q_0 and a compound action s_0 a trajectory $tr = \{(q_{i-1}, c_i, q_i, d_i)\}_{i \geqslant 1}$ from q_0 corresponds to such a path $\rho = \{(v_{i-1}, c_i, v_i)\}_{i \geqslant 1}$ in Γ_π that $v_0 = (q_0, s_0)$ and $v_i = (q_i, s_{i-1}h_i)$ for all $i \geqslant 1$. If $tr = \{(q_{i-1}, c_i, q_i, d_i)\}_{i=1}^k$ is an initial finite trajectory then we denote by $v_{init}[tr]$ such a node (q_k, h) of Γ_π that $h = h_1 h_2 \dots h_k$. If $\rho = \{(v_{i-1}, c_i, v_i)\}_{i \geqslant 1}$ is a path in Γ_π then we denote for every $k \geq 0$ by $\rho|^k$ its prefix $\{(v_{i-1}, c_i, v_i)\}_{i=1}^k$.

The verification of information processing systems is a checking that the actual behavior of a system satisfies the expected properties. By choosing finite state transducers for a formal model of sequential reactive system we thus formalize the notion of "behavior" of such systems: a behavior of a transducer π manifests itself in the set of trajectories $Tr_\pi(q_{init})$ of all initial runs of π; this set is represented by the computation tree Γ_π. Any set of trajectories can be regarded as a property of transducers behavior. It is well known that the properties of behaviors represented by infinite sequences of events, as well as discrete structures that combine these sequences, can be conveniently specified by means of temporal logics (LTL, CTL, CTL^*, etc.). However, when specification of transducers behavior is concerned, one should keep in mind that an adequate specification language must admit interpretation over dual sequences of input and output events. The authors of [13] drew attention to this particular feature of formal languages for specifying the behavior of transducers, and they proposed a novel logic $\mathcal{LP}$-LTL intended for reasoning about the behavior of transducers. This logic is an extension of LTL, where temporal operators are parametrized with languages over $\mathcal{C}$ and $\mathcal{A}$. In [13] it was shown that in the case when only regular languages are used as parameters of temporal operators the model checking of finite state transducers against *Reg-LTL* specifications is decidable in double exponential time. Next, in [6] *Reg-CTL* — a regular extension of *CTL* adapted for reasoning about dual sequences of events — was introduced, and it was proved that model checking problem for finite state transducers against *Reg-LTL* specifications is PSPACE-complete. The expressive power of these and some other extensions of temporal logics was studied in [7]. It is quite natural that the next stage of research is the study of the verification problem for the extension of even more general logic CTL^*. In this section we present a temporal logic $\mathcal{LP}$-CTL^* and its regular fragment *Reg*-CTL^*, which is of particular interest for model checking of finite state transducers.

As it was noticed above, the correct behavior of a sequential reactive system depends on how it responds to certain environmental requests. The type of environmental impact on the system can be represented as a language L over the set of input signals, and the type of the response to such a stimulus — as a language P over the set of actions. A language L over $\mathcal{C}$ is called an *environment behavior pattern*, and a language P over $\mathcal{A}$ is called a *basic predicate*.

Suppose that we are given a family $\mathcal{L}$ of environment behavior patterns and a family $\mathcal{P}$ of basic predicates $\mathcal{P}$. The set of $\mathcal{LP}$-CTL^* formulae consists of the subset of *state formulae* and the subset of *path formulae* which are defined as follows:

1) every basic predicate $P \in \mathcal{P}$ is a state formula;

2) if φ_1, φ_2 are state formulae then $\neg\varphi_1$ and $\varphi_1 \wedge \varphi_2$ are state formulae;

3) if ψ is a path formula then $\mathbf{A}\psi$ and $\mathbf{E}\psi$ are state formulae;

4) if φ is a state formula then φ is a path formula;

5) if ψ_1, ψ_2 are path formulae then $\neg\psi_1$ and $\psi_1 \wedge \psi_2$ are path formulae;

6) if $\varphi, \varphi_1, \varphi_2$ are path formulae, $c \in \mathcal{C}$, and $L \in \mathcal{L}$ then $\mathbf{X}_c\varphi$ and $\varphi_1\, \mathbf{U}_L\varphi_2$ are path formulae.

An intuitive meaning of $\mathcal{LP}$-CTL^* formulae is as follows. A basic predicate P holds whenever an output result computed so far by a reactive system is a compound action from the set P. A formula $\mathbf{A}\psi$ (or $\mathbf{E}\psi$) means that every (some) computation of a reactive system satisfies the requirement ψ. A formula $\mathbf{X}_c\varphi$ claims that a reactive system is able to receive the input signal c and its subsequent behavior satisfies the requirement φ. A formula $\varphi_1\, \mathbf{U}_L\varphi_2$ asserts that every time after receiving a sequence w of input signals such that $w \in L$, the behavior of a system satisfies the requirement φ_1 until, upon receipt of an input sequence from L, the behavior of the system meets the requirement φ_2.

Formally, $\mathcal{LP}$-CTL^* formulae are interpreted over computation graphs Γ_π of finite state transducers $\pi = (Q, \mathcal{C}, \mathcal{A}, q_{init}, T)$. By writing $\Gamma_\pi, v \models \varphi$ we indicate that a state formula φ is satisfied at the node v of Γ_π, and by writing $\Gamma_\pi, \rho \models \psi$ we indicate that a path formula ψ is satisfied on a path ρ of Γ_π. The satisfiability relation $\models$ is defined by structural induction on the formulae for every node $v = (q, s)$ in the computation graph Γ_π, and a path ρ in Γ_π assuming that $\rho = (v_0, c_1, v_1), (v_1, c_2, v_2), \dots$ as follows:

1. $\Gamma_\pi, v \models P \iff s \in P$ for every basic predicate $P \in \mathcal{P}$;
2. $\Gamma_\pi, v \models \neg\varphi \iff$ it is not true that $\Gamma_\pi, v \models \varphi$;
3. $\Gamma_\pi, v \models \varphi_1 \wedge \varphi_2 \iff \Gamma_\pi, v \models \varphi_1$ and $\Gamma_\pi, v \models \varphi_2$;
4. $\Gamma_\pi, v \models \mathbf{E}\,\varphi \iff$ there exists such a path ρ from the node v in Γ_π that $\Gamma_\pi, \rho \models \varphi$;
5. if φ is a state formula then $\Gamma_\pi, \rho \models \varphi \iff \Gamma_\pi, v_0 \models \varphi$;
6. $\Gamma_\pi, \rho \models \neg\psi \iff$ it is not true that $\Gamma_\pi, \rho \models \psi$;
7. $\Gamma_\pi, \rho \models \psi_1 \wedge \psi_2 \iff \Gamma_\pi, \rho \models \psi_1$ and $\Gamma_\pi, \rho \models \psi_2$;
8. $\Gamma_\pi, \rho \models \mathbf{X}_c\varphi \iff c = c_1$ and $\Gamma_\pi, \rho|^1 \models \varphi$;
9. $\Gamma_\pi, \rho \models \varphi\,\mathbf{U}_L\psi \iff \exists i \geqslant 0$: $c_1c_2\dots c_i \in L$ such that $\Gamma_\pi, \rho|^i \models \psi$ and $\forall j,\ 0 \leqslant j < i$, if $c_1c_2\dots c_i \in L$ then $\Gamma_\pi, \rho|^j \models \varphi$.

In the case when $v = v_{init}$ we will write $\pi \models \varphi$ instead of $\Gamma_\pi, v \models \varphi$. In the definition above environment behavior patterns and basic predicates may be arbitrary languages over the alphabet of signals and the alphabet of actions, respectively. As one might expect, this freedom of choice leads to undecidability. For example, let $\mathcal{C}$ be an alphabet of two or more letters, and $\mathcal{A} = \mathcal{C}$. Consider such a transducer π that $T = \{(q_{init}, c, c, q_{init}) : c \in \mathcal{C}\}$, i.e. π just retransmits the received signals. Then, for any pair of context-free languages U and V over $\mathcal{C}$ it is true that $\pi \models \mathbf{EF}_U V$ iff $U \cap V \neq \varnothing$. Since the emptiness of intersection problem for context-free languages is undecidable (see [10]), the problem of checking whether a transducer π satisfies a $\mathcal{LP}$-CTL^* formula φ (the *model checking problem* $\pi \models \varphi$) is also undecidable when $\mathcal{L}$ and $\mathcal{P}$ are classes of context-free languages.

In order to find an effective solution to the model checking problem for $\mathcal{LP}$-CTL^* thus introduced, we restrict ourselves to more simple classes of environment patterns and basic predicates. In Section 4 we consider a fragment of $\mathcal{LP}$-CTL^*, namely, *Reg*-CTL^*, where all environment behavior patterns and all basic predicates are *regular languages.* But first, it is worthwhile to briefly compare the logic $\mathcal{LP}$-CTL^* introduced here with other previously known extensions of conventional temporal logics.

3. Other extensions of temporal logics

In [5] the first comprehensive analysis of LTL as a formal language for describing the behavior of computing systems was carried out, and several attempts were made shortly thereafter to improve the expressive power of this temporal logic. The modifications were made mainly in two directions: 1) adding new expressive means (quantifiers, modalities, etc.), and 2) changing the semantics of temporal operators existing in LTL.

For example, the authors of [15] supplied the syntax of LTL with quantification for basic predicates and discovered that the expressive power of the quantified extension thus introduced significantly exceeds the descriptive capabilities of the plain LTL. On the other hand, in [20] a method for introducing new temporal operators using right-linear grammars was proposed. The words generated by these grammars define the patterns on which the satisfiability of the formulas in the scope of a temporal operator is checked. Similarly, in [14, 18] it was shown that the patterns for describing the semantics of new temporal operators can be defined by means of finite automata. The idea of supplying temporal operators with some parameters is not new: almost the same parameterization of temporal operators as in this paper was introduced in [9] for dynamically extend LTL. Since then several attempts of this kind were made to merge regular languages and temporal operators (e.g. see [16] among the latest). In almost all these cases a remarkable effect was found: an expressive power of such extensions increases and becomes equivalent to that of $S1S$ logic, while satisfiability checking problem for these logics remains PSPACE-complete.

When introducing $\mathcal{LP}$-CTL^*, we did not seek only to improve the expressive power of CTL^* as such. Our goal was to offer an adequate language for the specification of the behavior of reactive systems modeled by transducers. In the computation of a transducer, a coordinated formation of two sequences is carried out — a sequence of input signals and a sequence of output actions. Therefore, a distinctive feature of $\mathcal{LP}$-CTL^* semantics is a certain synchronization of the parameters of temporal operators (their interpretation is determined by the sequence of input signals) and the truth values of basic predicates (they depend on the sequence of output actions). It could be said that the semantics of our logic is defined on traces in two-dimensional space, while in all previously known parameterized extensions of temporal logics only one-dimensional traces were used. This feature significantly affects algorithms for checking the satisfiability of formulas.

4. Model checking of finite state transducers against *Reg*-CTL^* specifications

The study of model checking problem for finite state transducers was first initiated in [13]. The authors of this paper considered the case when specifications are given by *Reg-LTL* formulae. This logic is an extension of the well-known temporal logic LTL and it is also *linear* fragment of *Reg*-CTL^* which contains only the formulae of the type $\mathbf{A}\varphi$, where φ is a quantifier-free path formula. The model checking procedure developed in [13] is based on a translation of a pair (π, φ) to a Büchi automaton $B(\pi, \varphi)$ such that $\pi \models \varphi$ iff $B(\pi, \varphi) \neq \varnothing$.

In [6] the study of the model checking problem for finite state transducers was continued. The authors of this paper considered the temporal logic *Reg*-CTL. This logic is an extension of another well-known temporal logic CTL; and it is also another fragment of *Reg*-CTL^* which consists of those formulae where each temporal operator $\mathbf{F}, \mathbf{G}$ or $\mathbf{U}$ is preceded by a path quantifier $\mathbf{E}$ or $\mathbf{A}$.

Model checking algorithms for *Reg-LTL* and *Reg*-CTL developed in [6, 13] follow respectively the automata-theoretic and tableau-based approaches used for the solution of model checking problem for conventional temporal logics LTL and CTL. In [4] it was shown (see also [3]) that model checking problem for Extended Computational Tree Logic CTL^* can be solved with essentially the same complexity as LTL, using a combination of the algorithms for LTL and CTL. However, when model checking of transducers against *Reg*-CTL^* formulae is concerned some specific features of transducers behavior make it impossible a straightforward application of this combination techniques; model checking of *Reg*-CTL^* specifications of finite state transducers needs a far elaborate study. In this section we present a model checking algorithm for *Reg*-CTL^* which is based on an iterated translation of a *Reg*-CTL^* formula into Büchi automata.

4.1. Finite automata and Büchi automata

A deterministic *finite state automaton* (DFA) over an alphabet Σ is a quintuple $A = (Q, \Sigma, q_{init}, \delta, F)$, where Q is a finite set of states, $q_{init} \in Q$ is an initial state, $F \subseteq Q$ is a set of *accepting states* and $\delta\colon Q \times \Sigma \mapsto Q$ is a transition function. This function can be extended to the set of words Σ^* as follows: $\delta(q, \varepsilon) = q$, and $\delta(q, \sigma x) = \delta(\delta(q, \sigma), x)$ for all $q \in Q, \sigma \in \Sigma$ and $x \in \Sigma^*$. A DFA A *accepts* a word x (we will write $x \in A$ to denote this fact) iff $\delta(q_{init}, x) \in F$. A DFA A *recognizes* a language $L(A) = \{x : x \in A\}$ of all words it accepts.

A DFA $A[x] = (Q, \Sigma, \delta(q_{init}, x), \delta, F)$ is called a *shift* of a DFA A *by* a word x. The *size* $|A|$ of an automaton A is the size of a table description of its transition function δ. By the size $|L|$ of a regular language L we mean the size $|A|$ of the *minimal* DFA A which recognizes L.

In addition to deterministic automata, we will also use *nondeterministic* finite state automata (NFA) for manipulations with regular languages. NFA have transition functions of the type $Q \times \Sigma \mapsto 2^Q$, and its extension to Σ^* is defined by the equalities $\delta(q, \varepsilon) = \{q\}$, and $\delta(q, \sigma x) = \bigcup_{q' \in \delta(q,\sigma)} \delta(q', x)$. A NFA A accepts a word x iff $\delta(q^A_{init}, x) \cap F \neq \varnothing$ holds.

Finite state automata can be further extended to allow recognition of sets of *infinite* words over Σ. A nondeterministic generalized *Büchi automaton* over an alphabet Σ is a quintuple $B = (Q, \Sigma, q_{init}, \Delta, \mathcal{F})$, where Q is a finite set of states, q_{init} is an initial state, $\Delta \subseteq Q \times \Sigma \times Q$ is a *transition relation*, and $\mathcal{F} = \{F_1, \dots, F_m\}$ is an *accepting rule*, where $F_i \subseteq Q, 1 \leqslant i \leqslant m$. For every infinite word $x = \sigma_0\sigma_1\sigma_2 \cdots \in \Sigma^\omega$, a *run* of B on x is an infinite sequence $q_0, q_1, q_2, \dots$ of states of B, where $q_0 = q_{init}$ and for all $i, i \geqslant 0$, $(q_i, \sigma_i, q_{i+1}) \in \Delta$. A run $q_0, q_1, q_2, \dots$ is *accepting* if for all $i, 1 \leqslant i \leqslant m$, there exist infinitely many j, such that $q_j \in F_i$. A word $x \in \Sigma^\omega$ is *accepted* by B (we write $x \in B$ to denote this fact) iff there exists an accepting run of B on x. In [3] it was shown that there exists a linear-time algorithm for checking, given a Büchi automaton B, iff $B = \varnothing$.

4.2. Translation of *Reg-CTL** formulae to automata

Consider a finite state transducer $\pi = (Q, \mathcal{C}, \mathcal{A}, q_{init}, T)$ and a *Reg-CTL** formula φ. A *model checking automaton* is such a DFA $A(\pi, \varphi) = (Q_A, T, q^A_{init}, \delta_A, F_A)$ over the finite alphabet T of transitions of π which satisfies for any finite trajectory $tr \in \mathrm{Pref}(Tr_\pi(q_{init}))$ the following relationship: $\Gamma_\pi, v_{init}[tr] \models \varphi \iff tr \in A(\pi, \varphi)$.

Clearly, having constructed such a DFA $A(\pi, \varphi)$, we obtain a solution to the model checking problem, since $\pi \models \mathbf{E}\varphi$ iff $\varepsilon \in A(\pi, \varphi)$. The main result of the paper is

Theorem 1. For every finite state transducer π and a state *Reg-CTL**-formula φ the model-checking automaton $A(\pi, \varphi)$ can be effectively constructed within a memory space $|\pi| \cdot 2^{\mathrm{poly}(|\varphi|)}$.

Proof. From the definition of satisfiability relation for *Reg-CTL** formulae it follows that $\models P \equiv \mathbf{E}P$ holds for every basic predicate P. Therefore, it may be assumed without loss of generality that every occurrence of a basic predicate in φ follows a path quantifier $\mathbf{E}$.

Next, it should be noted that every state *Reg-*CTL^*-formula φ can be attributed to one of three types of formulas:

1. $\varphi = \mathbf{E}P$ for some basic predicate P;
2. $\varphi = \Phi(\mathbf{E}\varphi_1, \dots, \mathbf{E}\varphi_m)$ for some Boolean formula $\Phi(p_1, \dots, p_m)$;
3. $\varphi = \mathbf{E}\psi(\mathbf{E}\varphi_1, \dots, \mathbf{E}\varphi_m)$ for some quantifier-free path *Reg-*CTL^*-formula $\psi(P_1, \dots, P_m)$.

Therefore, the construction of the model checking automaton $A(\pi, \varphi)$ is carried out recursively.

1. For every regular basic predicate P the model checking automaton $A(\pi, \mathbf{E}P)$ can be easily built of a transducer π and a DFA A_P which recognizes P.
2. For every Boolean formula $\Phi(p_1, \dots, p_m)$ a model checking automaton $A(\pi, \varphi)$ for a state *Reg-*CTL^*-formula $\varphi = \Phi(\mathbf{E}\varphi_1, \dots, \mathbf{E}\varphi_m)$ can be built as a Boolean combination of model checking automata $A(\pi, \mathbf{E}\varphi_i)$ corresponding to subformulae $\mathbf{E}\varphi_i, 1 \le i \le m$.
3. Suppose that $\varphi = \mathbf{E}\psi(\mathbf{E}\varphi_1, \dots, \mathbf{E}\varphi_m)$, where $\psi(P_1, \dots, P_m)$ is some quantifier-free path *Reg-*CTL^*-formula, and model checking automata $A(\pi, \mathbf{E}\varphi_i), 1 \le i \le m$, are available. Then regular languages $L(A(\pi, \mathbf{E}\varphi_i))$ recognized by these automata are regarded as basic predicates $P_1, \dots, P_m$, and model-checking automaton $A(\pi, \varphi)$ is built as follows:
   - by applying a model checking algorithm proposed in [13] build a Büchi automaton $B_{\psi(P_1,\dots,P_m)}$ which accepts a trace $tr \in Tr_\pi$ iff tr corresponds to such a path ρ that $\Gamma_\pi, \rho \models \psi(P_1, \dots, P_m)$;
   - next, by combining a Büchi automaton $B_{\psi(P_1,\dots,P_m)}$ and a transducer π build such a Büchi automaton $B(\pi, \varphi)$ that for any finite trajectory tr the following relationship holds: $\Gamma_\pi, v_{init}[tr] \models \mathbf{E}\psi(P_1, \dots, P_m)$ iff there exists an accepting run of $B(\pi, \varphi)$ on some trace tr' which is an extension of tr;
   - finally, by using a simple reachability checking techniques build a model checking automaton $A(\pi, \varphi)$ from $B(\pi, \varphi)$.

Corollary. The model checking problem for finite state transducers against *Reg-*CTL^* specifications is in EXPSPACE.

Theorem 2. The model checking problem for finite state transducers against *Reg-*CTL^* specifications is PSPACE-hard.

Proof (sketch). In [12] it was proved that it is PSPACE-hard to check whether the intersection of an arbitrary number of deterministic finite state automata is empty. This problem can be reduced to our model checking problem in polynomial time. □

5. Conclusion

In this paper we introduced an extension *Reg-CTL** of Generalized Computational Tree Logic CTL^* as a formal language for specification the behavior of sequential reactive systems, define its semantics on the models of finite state transducers, and give a solution to the model checking problem for finite state transducers against formulae from *Reg-CTL**. The solution is obtained by means of automata-theoretic techniques following the ideas of our model checking algorithms for and *Reg-LTL* and *Reg-CTL* presented in the early papaers [8**?**]. It may be noticed that this model checking algorithm is exponentially more time consuming than the similar algorithm for plain CTL^* (see [**?**]). However, there remains an exponential gap between the lower and the upper bounds on the complexity of the model checking problem for *Reg-CTL**. We assume that this gap can be filled by improving the proposed algorithm using the promising techniques suggested in [9], where a logic similar to *Reg-LTL* was studied.

The authors of the article are grateful to the anonymous reviewers for useful comments that helped to improve the article.

Список литературы

1. Гнатенко А. Р., Захаров В. А. О сложности верификации автоматов-преобразователей над коммутативными полугруппами // Материалы XVIII Международной конференции “Проблемы теоретической кибернетики”. 2017. С. 68–71.
2. Alur R., Cerny P. Streaming transducers for algorithmic verification of single-pass list-processing programs // Proceedings of the 38-th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. 2011. P. 599–610.
3. Clarke E.M., Gramberg O., Kroening D., Peled D.A., Veith H. Model Checking. MIT Press. 2018. 424 p.
4. Emerson E.A., Lei C.-L. Modalities for model checking: branching time logic strikes back // Science of Computer Programming. 1987. Vol. 8, № 3. P. 275–306.
5. Gabbay D., Pnueli A., Shelach S., Stavi J. The temporal analysis of fairness // Proceedings of the 7-th ACM Symposium on Principles of Programming Languages. 1980. P. 163–173.
6. Gnatenko A.R., Zakharov V.A. On the model checking of finite state transducers over semigroups // Proceedings of ISP RAS. 2018. Vol. 30, № 3, P. 303–324
7. Gnatenko A.R., Zakharov V.A. On the expressive power of some extensions of Linear Temporal Logic // Automatic Control and Computer Sciences. 2019. Vol. 53, № 7, P. 506–524.
8. Gnatenko A.R. On the complexity of model checking problem for finite state transducers over free semigroups. // Proceedings of the Student Session of European Summer School on Logic, Language and Information, Riga, 2019.

9. Henriksen J.J., Thiagarajan P.S., Dynamic linear time temporal logic // Annals of Pure and Applied Logic. 1999. Vol. 96, № 1–3. P. 187–207.
10. Hopcroft J.E., Ullman J.D. Introduction to Automata Theory, Languages, and Computation (1st ed.). Addison-Wesley. 1979.
11. Hu Q., D'Antoni L. Automatic Program Inversion using Symbolic Transducers // Proceedings of the 38-th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2017. P. 376–389.
12. Kozen D. Lower bounds for natural proof systems. // Proceedings of the 18-th Symposium on the Foundations of Computer Science. 1977. P. 254–266.
13. Kozlova D.G., Zakharov V.A. On the model checking of sequential reactive systems // Proceedings of the 25th International Workshop on Concurrency, Specification and Programming (CS&P 2016), CEUR Workshop Proceedings. 2016. Vol. 1698, P. 233–244.
14. Kupferman O., Piterman N., Vardi M.Y. Extended temporal logic revisited // Proceedings of the 12-th International Conference on Concurrency Theory. 2001. P. 519–535.
15. Manna Z., Wolper P. Synthesis of communicating processes from temporal logic specifications // ACM Transactions on Programming Languages and Systems. 1984. Vol. 6, № 1. P. 68–93.
16. Mateescu R., Monteiro P.T., Dumas E., De Jong H. CTRL: Extension of CTL with regular expressions and fairness operators to verify genetic regulatory networks // Theoretical Computer Science. 2011. Vol. 412, № 26. P. 2854–2883.
17. Savitch W.J. Relationships between nondeterministic and deterministic tape complexities. // Journal of Computer and System Sciences, 1970. Vol. 4, № 2.
18. Vardi M.Y., Wolper P. Yet another process logic // Proceedings of the Carnegie Mellon Workshop on Logic of Programs. 1983. P. 501–512.
19. Veanes M., Hooimeijer P., Livshits B., et al. Symbolic finite state transducers: algorithms and applications // Proceedings of the 39-th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ACM SIGPLAN Notices. 2012. Vol. 147. P. 137–150.
20. Wolper P. Temporal logic can be more expressive // Information and Control. 1983. Vol. 56, №№ 1–2, P. 72–99.

UDK 004

Reasoning about Programmable Logic Controllers

Garanina N.O. (Institute of Informatics Systems SB RAS, Institute of Automation and Electrometry SB RAS)

Anureev I.S. (Institute of Informatics Systems SB RAS, Institute of Automation and Electrometry SB RAS)

Zyubin V.E. (Institute of Automation and Electrometry SB RAS)

Rozov A.S. (Institute of Automation and Electrometry SB RAS)

Liakh T.V. (Institute of Automation and Electrometry SB RAS)

Gorlatch S.(University of Muenster)

We address the formal verification of the control software of critical systems, i.e., ensuring the absence of design errors in a system with respect to requirements. Control systems are usually based on industrial controllers, also known as Programmable Logic Controllers (PLCs). A specific feature of a PLC is a scan cycle: 1) the inputs are read, 2) the PLC states change, and 3) the outputs are written. Therefore, in order to formally verify PLC, e.g., by model checking, it is necessary to reason both in terms of state transitions within a cycle and in terms of larger state transitions according to the scan-cyclic semantics.

We develop a formalization of PLC as a hyperprocess transition system and an LTL-based temporal logic cycle-LTL for reasoning about PLC.

Keywords: *formal verification, model checking, temporal logics, transition systems, programmable logic controllers*

1. Introduction

The long-term goal of our work is formal verification of control software specified in the process-oriented paradigm, in particular programs written in the Domain-Specific Language (DSL) Reflex [1, 7]. Formal verification of Programmable Logic Controllers (PLCs), which are important components of automatic control systems, is an active topic of practical and research work [2, 4]. The proposed approaches follow various PLC models [8]. In general, PLC functioning consists of infinite sequence of *scan cycles*. Each scan cycle includes a sequence of three phases: reading input, execution, and writing output.

We choose a process-oriented PLC modeling based on *hyperprocesses* [7]. This modeling method allows us to specify the main features of the PLC, such as scan cycle and timers; it describes a PLC as a synchronized system of interacting functional processes defined by a set of functional states and actions on these states. According to the classification [8], hyperprocesses model PLCs that abstract from scan cycle time (the environment is considered to be slow enough to assume zero time for the input/output and execution phases of a scan cycle) and use input and output controlling timers. This modeling provides a natural specification for the control multiprocess systems, which, due to abstracting from the scan cycle time, allows ordering process executions in a scan cycle. Such high synchronization of processes without interleaving makes possible the effective use of formal verification methods. A hyperprocess is a base for the process-oriented language Reflex that was used in a number of industrial projects, in particular, a plant for growing silicon single crystals using Czochralski method, and a vacuum system for the Big Solar Vacuum Telescope [6].

In this paper, we assume model checking as our formal verification method. Therefore, we present a hyperprocess as a special transition system, and the properties of a hyperprocess as formulas of a special temporal logic. A hyperprocess transition system is close to a concurrent multi-threaded system [3] enriched with synchronizing counter, process functional states, timers, and action primitives for their changing. The time for a PLC specified as a hyperprocess is clocked both inside of the scan cycle execution (taking into account changes of variable values) and outside of the execution at the reading of the inputs.

The logic for reasoning about PLC should allow the formulation of statements for these two kinds of clocks. In this paper, we develop the *cycle-LTL* logic – which is an LTL enriched with cycle temporal operators for reasoning about PLC states outside of a scan cycle. In this logic, the following properties are expressible for a simple example of a hand-dryer machine: "*If the sensor has detected hands, the dryer will turn on in the next scan cycle*" or "*If the temperature is higher than the critical value, the cooling process is always on.*" Note that if the switching on and off for the cooling process is performed after actions of other processes in a scan cycle, then the last property can be violated inside the cycle, but it can hold outside the cycle. This example illustrates the need to use cyclic temporal operators, in particular special cycle always-operator $\mathbf{G^c}$.

2. Hyperprocess Transition System

Let us give an informal description of the hyperprocess [7]. *Hyperprocess* is an ordered set of interacting processes that execute sequentially in a given order, forming a *scan cycle.* This cycle starts by reading the input from the environment into the hyperprocess system variables, and finishes by writing outputs to the environment. All hyperprocess variables are global. A feature of the processes that form a hyperprocess are *functional states* – labels that mark a sequence of process actions. The functional states of every process include the states of normal shutdown and abnormal shutdown: in these states, the process does not perform any actions. Process actions can change the values of hyperprocess variables (except input variables), affect the functional states of other processes, and set and reset timer values. Actions may have guard conditions depending on the hyperprocess variables and functional states of other processes. Our definition of the hyperprocess transition system is based on a description of the hyperprocesses and the operational semantics of the Reflex language [1, 7]. We hide the output phase inside the execution phase without loss of generality.

Definition 1. *(Hyperprocess transition systems, HTS)*
HTS is a tuple $H = (P, S, s_{ini}, A, R)$, where

- $P = \{p_1, ..., p_n\}$ is an ordered set of processes;
- S is a nonempty set of states;
- s_{ini} is an initial state;
- A is an action alphabet;
- R is a labelled transition relation $R : A \mapsto 2^{S\times S}$.

Before defining HTS-components, we describe hyperprocess elements in general.

Definition 2. *(Hyperprocess elements)*
Hyperprocess elements are variables, functional states, process actions, and timers:

Variables. $V = \{v_1, \dots, v_N\}$ is a set of hyperprocess variables which values are the result of the corresponding functions $v_i : S \mapsto D \cup \{\bot\}$. We distinguish *input variables* V_E and *process variables* V_P: $V = V_E \cup V_P$.

Functional states. For every $i \in [1..n]$ $F_i = \{f_i^1, ..., f_i^{m_i}, stop, err\}$ is *a set of functional states* of process p_i. The *stop* and *err* are *inactive* states, and other states are *active* states. The value of every functional state variable f_i is described by function $f_i : S \mapsto F_i$.

Actions. In functional state f_i^j, process p_i performs actions from set A. These actions form *the body of the functional state.* $L_i^j \in \mathbb{N}$ is the number of actions in this body. Variable a_i is *an action counter* and its value is *a position.* The value of the action counter is the result of function $a_i : S \mapsto [1..L_i^j] \cup \{\bot\}$. The next value of this counter is defined by functions $nxt^j : \mathbb{N} \times S \mapsto \mathbb{N} \cup \{\bot\}$. These functions implicitly include guards for actions because the results depend on a current HTS-state, in particular, on activity of other processes. If there is no the next action in the current functional state then $nxt^j(a_i) = \bot$. Functions $an^j : \mathbb{N} \cup \{\bot\} \mapsto A$ return the name of the action at position a_i.

Timers. *The timer* of process p_i is variable t_i with values as the result of functions $t_i : S \mapsto \mathbb{N} \cup \{\bot\}$. *The timer bound* at position a of functional state f_i^j is $N_i^{ja} \in \mathbb{N}$. For simplicity, we consider the processes with one timer per a functional state.

Let us define the components of HTS-system $H = (P, S, s_{ini}, A, R)$.

The set of states S

A state $s = (v, sp, pc) \in S$ includes the following elements:

- the state of variables $v = (v_1(s), \dots, v_N(s))$ for variables' values in state s;
- the state of processes $sp = ((f_1(s), a_1(s), t_1(s)), \dots, (f_n(s), a_n(s), t_n(s)))$, where for every process p_i in state s, $f_i(s)$ is its current functional state, $a_i(s)$ is an action counter in state $f_i(s)$, and $t_i(s)$ is the value of its timer;
- process counter $pc(s)$ with values in $[0..n]$, where 0 is reserved for updating input.

The initial state s_{ini}

$s_{ini} = (v_0, sp_0, pc_0)$, where

– $v_0 = (\bot_1, \dots, \bot_N)$,

– $sp_0 = ((f_1^1, 1, \bot)_1, (stop, \bot, \bot)_2, \dots, (stop, \bot, \bot)_n)$, and

– $pc_0 = 0$.

The action alphabet A

The alphabet includes a single environment action and process actions:

$A = \{upd0, skip, end, upd, tout, reset, startP, stopP, start, set, next, stop, err\}$.

0. The cycle action.

*upd*0 – change of values of input variables, i.e. reading environment inputs.

1. Service actions.

skip – a process does nothing in inactive states.

end – a process transfers control to the next process at the end of its active state body.

2. Actions for updating non-input variables.

upd – a process changes the values of some variables.

3. Timeout actions.

tout – a process starts the timer and performs actions in the current state until timeout;

reset – a process resets its timer to zero.

4. Actions for functional states.

startP/stopP – a process transfers target process p_k to functional state $f_k^1/stop$;

start/set – a process transits to target functional state f_i^1/f;

stop/err – a process transits to functional state $stop/err$;

next – a process transits to the next functional state.

The labeled transition relation R

The transition relation R gives the semantics to actions of processes and the environment. Let $i \in [1..n], j \in [1..m_i]$, $a \in [1..L_i^j]$. We use the following notation.

The expression

$$(f_i = f_i^j, a_i = a, T_i, Tg_i, pc = i) \xrightarrow{act} (y_1' = new_1, \dots, y_{m'}' = new_{m'})$$

means that $R(act) = (s, s')$, where

- $act = an^j(a)$, and $an^j(\perp) \in \{skip, end\}$;
- before-state s is such that $pc(s) = i$, $f_i(s) = f_i^j$, $a_i(s) = a$, the time constraint T_i can be $t_i(s) \neq \perp$, $t_i(s) = \perp$, $t_i(s) < N_i^j$, or $t_i(s) = N_i^j$, and the target constraint Tg_i can be $tgp_i = k$ or $tgs_i = k$, where tgp_i is variable in V_P with values in $[1..n]$ for specifying the target process, and tgs_i is variable in V_P with values in $[1..m_i]$ for specifying the target functional state of process p_i; non-mentioned left elements have arbitrary values;
- after-state s' specifies the changes of hyperprocess elements after action act: $y_k(s') = new_k$ ($k \in [1..m']$); non-mentioned right elements are not changed.

0. The external update action.

We use the notation like above to specify $R(upd0)$. At the beginning of the scan cycle, the values of input data are read to the input variables in V_E, the value of every ticking process timer is increased by 1, and the process counter points to the first process:

$(t_{i_1} \neq \perp, \dots, t_{i_k} \neq \perp, pc = 0) \xrightarrow{upd0}$

$$(v_1' = v_1, \dots, v_m' = v_m, t_{i_1}' = t_{i_1} + 1, \dots, t_{i_k} = t_{i_k} + 1, pc' = 1).$$

In the following definition of the transition relation R for process actions, we suppose that process p_i performs the action number a_i with name $an^j(a_i)$ in its functional state $f_i = f_i^j$.

1. Service actions.

Process p_i does nothing in inactive states *stop* and *err*, and passes control to the next process:

– $(f_i = stop, pc = i) \xrightarrow{skip} (pc' = |i+1|_{n+1})$.

– $(f_i = err, pc = i) \xrightarrow{skip} (pc' = |i+1|_{n+1})$.

When process p_i reaches the end of the body of its active functional state f_i^j, it goes to the first action of the body and transfers control to the next process:

– $(f_i = f_i^j, a_i = \bot, pc = i) \xrightarrow{end} (a_i' = 1, pc' = |i+1|_{n+1})$.

2. The action for updating variable values.

By this action, process p_i changes values of some variables from the set of non-input variables $\{v_1, \dots, v_m\} \subseteq V_P$, and goes to the next action $nxt^j(a)$ of its current state:

– $(f_i = f_i^j, pc = i) \xrightarrow{upd} (v_1' = d_1, \dots, v_m' = d_m, a_i' = nxt^j(a))$;

3. Timeout actions.

In these cases, process p_i starts the timer t_i (if $t_i = \bot$), goes to the first action of its current state f_i^j, and transfers control to the next process:

– $(f_i = f_i^j, t_i = \bot, pc = i) \xrightarrow{tout} (a_i' = 1, t_i' = 0, pc' = |i+1|_{n+1})$;

– $(f_i = f_i^j, t_i < N_i^j, pc = i) \xrightarrow{tout} (a_i' = 1, pc' = |i+1|_{n+1})$.

In the case of timeout, process p_i stops the timer t_i and goes to the next action in its current functional state:

– $(f_i = f_i^j, t_i = N_i^j, pc = i) \xrightarrow{tout} (a_i' = nxt^j(a), t_i' = \bot)$.

The *reset*-action results exactly as the first case of the *tout*-action:

– $(f_i = f_i^j, t_i \neq \bot, pc = i) \xrightarrow{reset} (a_i' = 1, t_i' = 0, pc' = |i+1|_{n+1}))$.

4. Actions for functional states.

These two actions of process p_i force process p_k $(i \neq k)$ to go to start or stop state:

– $(f_i = f_i^j, tgp_i = k, pc = i) \xrightarrow{startP} (f_k' = f_k^1, a_k' = 1, t_k' = \bot, a_i' = nxt^j(a))$;

– $(f_i = f_i^j, tgp_i = k, pc = i) \xrightarrow{stopP} (f_k' = stop, a_k' = \bot, t_k' = \bot, a_i' = nxt^j(a))$;

With these actions, process p_i goes to the first action of the corresponding functional states and stops the timer:

– $(f_i = f_i^j, tgs_i = k, pc = i) \xrightarrow{set} (f_i' = f_i^k, a_i' = 1, t_i' = \bot)$;

– $(f_i = f_i^j, pc = i) \xrightarrow{start} (f_i' = f^1, a_i' = 1, t_i' = \bot)$;

– $(f_i = f_i^j, pc = i) \xrightarrow{next} (f_i' = f_i^{j+1}, a_i' = 1, t_i' = \bot)$;

Process p_i perform these actions in case of it should do nothing from this moment because of normal or error shutdown:

– $(f_i = f_i^j, pc = i) \xrightarrow{stop} (f_i' = stop, a_i' = \bot, t_i' = \bot)$;

– $(f_i = f_i^j, pc = i) \xrightarrow{err} (f_i' = err, a_i' = \bot, t_i' = \bot)$.

According to the defined transition relation, the processes act sequentially in the current cycle, following the order specified by the process counter. Let an input state s_{inp} be a state with $pc(s_{inp}) = 0$. We define *a cyclic state* s_c as a state just after updating the input variables and before the processes start to act: $R(upd) = (s_{inp}, s_c)$. The set of cycle states is S_c.

We define two kinds of paths in HTS. *Standard path* $\pi = s_0, s_1, \ldots$ is a sequence of states $s_i \in S$ such that $\forall i \geq 0\ \exists a \in A : R(a) = (s_i, s_{i+1})$. Let $\pi(i)$ be i^{th} state on path π. *Cycle path* $\sigma = c_0, c_1, \ldots$ is an infinite sequence of cycle states $c_i \in S_c$ such that for every $i \geq 0$ there exists a finite standard path π_i of length n_i with $\pi_i(0) = c_i$ and $\pi_i(n_i) = c_{i+1}$. If ρ is the standard or cycle path, let $\rho(k)$ be k^{th} state on this path, ρ^k be the suffix of ρ starting from $\rho(k)$, and c^π be the number of the first cycle state on standard path π.

3. Temporal logic cycle-LTL

The syntax of our cycle-LTL logic includes propositions P, boolean connections, standard LTL temporal operators, inner-cycle temporal operators, and cycle temporal operators:

$$\varphi ::= P \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{X}\varphi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi \mid \varphi\mathbf{U}\varphi \mid \\ \mathbf{X^i}\varphi \mid \mathbf{F^i}\varphi \mid \mathbf{G^i}\varphi \mid \varphi\mathbf{U^i}\varphi \mid \mathbf{X^c}\varphi \mid \mathbf{F^c}\varphi \mid \mathbf{G^c}\varphi \mid \varphi\mathbf{U^c}\varphi$$

The inner-cycle operators $\mathbf{X^i}$, $\mathbf{F^i}$, $\mathbf{G^i}$, and $\mathbf{U^i}$ are used for formulating properties of a control system which have to hold during a particular cycle execution phase, while *the cycle operators* $\mathbf{X^c}$, $\mathbf{F^c}$, $\mathbf{G^c}$, and $\mathbf{U^c}$ are used for formulating properties of the control system which have to hold at the beginning of cycles.

Let the set of standard LTL formulas be Φ^s, the set of formulas started with inner-cycle operators be Φ^i, the set of formulas started with cycle operators be Φ^c, and the set of all cycle-LTL formulas be Φ^{sic}.

The semantics of the cycle-LTL are defined for new inner-cycle and cycle temporal operators only. We define the semantics of inner-cycle operators on standard paths and semantics of cycle operators on standard and cycle path. The semantics of standard LTL formulas can be found in [3]. Let H be a hyperprocess transition system, π be an infinite standard path, σ be a cycle path, and $\varphi, \psi \in \Phi^{sic}$ be formula of cycle-LTL..

The semantics of formulas in Φ^i.

- $H, \pi \models \mathbf{X^i}\varphi$ iff $\pi^1 \notin S_c$ and $H, \pi^1 \models \varphi$;
- $H, \pi \models \mathbf{F^i}\varphi$ iff there exists $0 \leq k < c^\pi$ such that $H, \pi^k \models \varphi$;

- $H, \pi \models \mathbf{G^i}\varphi$ iff for all $0 \leq k < c^\pi$ $H, \pi^k \models \varphi$;
- $H, \pi \models \varphi\mathbf{U^i}\psi$ iff there exists $0 \leq k < c^\pi$ such that $H, \pi^k \models \psi$ and for all $0 \leq j < k$ $H, \pi^j \models \varphi$.

The semantics of formulas in Φ^c.

Let $\xi \in \Phi^c$.

- $H, \pi \models \xi$ iff $H, \sigma \models \xi$ with $\sigma(0) = \pi(c^\pi)$;
- $H, \sigma \models \mathbf{X^c}\varphi$ iff $H, \sigma^1 \models \varphi$;
- $H, \sigma \models \mathbf{F^c}\varphi$ iff there exists $k \geq 0$ such that $H, \sigma^k \models \varphi$;
- $H, \sigma \models \mathbf{G^c}\varphi$ iff for all $k \geq 0$ $H, \sigma^k \models \varphi$;
- $H, \sigma \models \varphi\mathbf{U^c}\psi$ iff there exists $k \geq 0$ such that $H, \sigma^k \models \psi$ and for all $0 \leq j < k$ $H, \sigma^j \models \varphi$.

Let us give some informal comments for semantics of cycle-LTL formulas. We consider the cases when a formula of some type is a subformula of other type formula at the first nesting level of temporal operators: $\varphi \in nl^1(\psi)$. Let $\varphi^s, \psi^s \in \Phi^s$, $\varphi^i, \psi^i \in \Phi^i$, and $\varphi^c, \psi^c \in \Phi^c$. We have six cases.

1. $\varphi^i \in nl^1(\psi^c)$: this assertion states that φ^i holds during the execution part of scan cycles explicitly specified by ψ^c.
2. $\varphi^i \in nl^1(\psi^s)$: this assertion states that φ^i holds during the execution part of scan cycles implicitly specified by ψ^s.
3. $\varphi^c \in nl^1(\psi^i)$: if φ^c is in boolean connection with formulas in $\Phi^s \cup \Phi^i$, this assertion binds a property of the execution phase of a scan cycle explicitly specified by ψ^i to a cycle property of the next cycle.
4. $\varphi^c \in nl^1(\psi^s)$: if φ^c is in boolean connection with formulas in $\Phi^s \cup \Phi^i$, this assertion binds a property of the execution phase of a scan cycle implicitly specified by ψ^s to a cycle property of the next cycle.
5. $\varphi^s \in nl^1(\psi^c)$: this assertion states that φ^s holds at some cycle state explicitly specified by ψ^c. If ψ^c include operator $\mathbf{G^c}$ or $\mathbf{U^c}$, φ^s will hold periodically with respect to scan cycle.
6. $\varphi^s \in nl^1(\psi^i)$: this assertion states that φ^s holds at some state in execution part of a scan cycle specified by ψ^i.

As an illustration, let us define the cycle-LTL specifications for the properties of a hand-dryer and cooling machine as example control systems:

1. *If the sensor has detected hands, the dryer will turn on in the next scan cycle*:

$\mathbf{G^c}(hands = on \to \mathbf{X^c} dryer = on)$.

2. *If the temperature is higher than the critical value, the cooling is always on*:

   $\mathbf{G^c}(temp > 95^\circ \to cooler = on)$.

The above examples of formulas for control system properties use only cycle states and observable input-output system variables. This formulation can be used for high-level properties of implementation independent models of control system viewed as a black box. However, if some high-level property fails for some implementation of the control system, then checking low-level properties of the system may be required. These properties are formulated in terms of process states and actions. They are hypotheses which use inner-cycle operators to localize the error within the scan cycle. Verifying such hypotheses can be less time-consuming then analyzing a counterexample for the failed high-level property. The following properties for a system that combines a lighting control system and a burglar alarm system illustrates causality between a low-level hypothesis and a high-level property: failing the former implies failing the latter.

1. *When a break-in is detected, all lamps should flash*:

   $\mathbf{G^c}(alarm \to \mathbf{X^c} alarm\_light = on)$.

2. *When the alarm sensor is on then the security alarm subsystem should send the alarm message to all other subsystems ASAP*:

   $\mathbf{G^c}(alarm \to \mathbf{F^i} alarm\_message\_sent)$.

4. Conclusion

In this paper, we develop the hyperprocess transition systems (HTS) for modeling PLC and the novel cycle-LTL logic for specifying PLC properties. Our HTS-model naturally captures features of PLC such as scan cycles and timers. Our cycle-LTL temporal logic enables reasoning about PLC properties w.r.t. both small-step time inside scan cycles and big-step time over scan cycles. Expressing the big-step properties in a standard LTL would be much more cumbersome.

We plan to prove that the model checking for HTS and cycle-LTL is reduced to the standard LTL model checking. For this we will translate HTS into the Kripke structure and cycle-LTL formulas into LTL formulas. The method of this translation will provide the basis for the correct translation of the process-oriented language Reflex into the Promela language used by the SPIN verifier [5]. We also plan to develop and implement a special model checking algorithm for verifying cycle-LTL formulas in HTS, which will have lower time complexity than the standard model-checking algorithm for LTL due to the use of HTS features such as cyclicity and ordering action processes.

References

1. Anureev I.S. Operational Semantics of Reflex // System Informatics. 2019. V. 14. P. 1–10.
2. Brinksma E., Mader A. Verification and Optimization of a PLC Control Schedule // In: Havelund K., Penix J., Visser W. (eds) SPIN Model Checking and Software Verification. SPIN. Springer, LNCS. 2000. V. 1885. P. 73–92.
3. Clarke E.M., Henzinger Th.A., Veith H., Bloem R. // Handbook of Model Checking. Springer International Publishing. 2018. Ch. 18.
4. Gourcuff V., de Smet O., Faure J.-M. Improving large-sized PLC programs verification using abstractions. // IFAC Proceedings. 2008. V. 41. № 2. P. 5101–5106.
5. Holzmann G. The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, Boston, 2003.
6. Kovadlo P.G., Lubkov A.A., Bevzov, A.N. et al. Automation system for the large solar vacuum telescope. // Optoelectronics, instrumentation and data processing. 2016. V. 52, P. 187–195.
7. Liakh T.V., Rozov A.S., Zyubin V.E. Reflex Language: a Practical Notation for Cyber-Physical Systems // System Informatics. 2018. V. 12. P. 85–104.
8. Mader A. A Classification of PLC Models and Applications. // In: Boel R., Stremersch G. (eds) Discrete Event Systems. SECS. Springer, Boston, MA, 2000. V. 569. P. 239–246.

УДК 004.05

Верификация программы преобразования строки в целое число

Шелехов В.И. (Институт систем информатики СО РАН, Новосибирский государственный университет)

Описывается дедуктивная верификация программы kstrtoul на языке Си из библиотеки ОС Linux. Программа kstrtoul реализует вычисление целого числа, представленного в виде последовательности литер. Чтобы упростить верификацию, применяются трансформации замены операций с указателями эквивалентными действиями без указателей. Программа преобразуется на язык предикатного программирования. Конструируется модель внутреннего состояния программы как часть спецификации программы. Дедуктивная верификация проведена в системах Why3 и Coq.

Ключевые слова: *дедуктивная верификация, трансформации программ, функциональное программирование, предикатное программирование, строковый тип.*

1. Введение

Исходной задачей является дедуктивная верификация программы kstrtoull.c на языке Си из библиотеки ядра ОС Linux. Программа kstrtoull.c реализует вычисление целого числа, представленного в строке в виде последовательности литер по правилам языка Си.

Дедуктивная верификация намного проще и быстрее для функциональных программ, чем для аналогичных императивных программ. Причина сложности императивных программ в том, что указатели, конструкции необходимые для оптимизации программ, существенно усложняют логику императивных программ. Для упрощения программ применяются трансформации, устраняющие указатели в императивной программе [6]. Операции с указателями заменяются эквивалентными действиями без указателей. Полученная программа преобразуется в эквивалентную предикатную программу. В системах автоматического доказательства Why3[18] и Coq [11] реализуется процесс дедуктивной верификации предикатной программы.

Предыдущий релиз верификации kstrtoull.c оказалася неудачным. Естественный и привычный стиль спецификации приводит к трудно доказуемым формулам корректности. Верификация в системе Why3 оказалась тяжелой. Далее, для упрощения верификации была разработана модель внутреннего состояния исполняемой программы. Из предикатной программы в максимальной степени экстрагирована ее константная часть. Использование модели в спецификации программы существенно упростило процесс дедуктивной верификации.

Во втором разделе настоящей работы дается краткое описание языка предикатного программирования. В третьем разделе описывается трансформация исходной программы kstrtoull с получением эквивалентной предикатной программы. В четвертом разделе определяется модель программы kstrtoull и детально описывается процесс спецификации предикатной программы с использованием модели. Далее строятся формулы корректности программы относительно спецификации применением системы правил [1, 7]. Построение формул корректности документируется в Приложении 3. Совокупность формул корректности вместе с описаниями типов и переменных оформляется в виде набора теорий. В Приложении 1 представлены теории с формулами корректности. Эти теории транслируются на язык спецификаций why3 [18]. В Приложении 2 приведены теории на языке Why3 для доказательства формул корректности на момент завершения работы по верификации. Особенности процесса дедуктивной верификации предикатной программы в системах Why3[18] и Coq [11] описывается в пятом разделе. В шестом разделе представлен обзор работ. Итоги верификации подводятся в седьмом разделе.

2. Язык предикатного программирования

Полная предикатная программа состоит из набора рекурсивных *предикатных программ* на языке P [2] следующего вида:

```
<имя программы>(<описания аргументов>: <описания результатов>)
pre <предусловие>
post <постусловие>
measure <выражение>
{ <оператор> }
```

Предусловие и постусловие являются формулами на языке исчисления предикатов. Они обязательны при дедуктивной верификации [3, 4, 8, 9, 15]. Мера задается только для рекурсивных программ и используется для доказательства их завершения.

Ниже представлены основные конструкции языка P: оператор присваивания, блок (оператор суперпозиции), параллельный оператор, условный оператор, вызов программы и описание переменных, используемое для аргументов, результатов и локальных переменных.

```
<переменная> = <выражение>
{<оператор1>; <оператор2>}
<оператор1> || <оператор2>
if (<логическое выражение>) <оператор1> else <оператор2>
<имя программы>(<список аргументов>: <список результатов>)
<тип> <пробел> <список имен переменных>
```

Всякая переменная характеризуется *типом* – множеством допустимых значений. *Описание типа* **type** T(p) = D с возможными параметрами p связывает имя типа T с его изображением D. Типы **bool**, **int**, **real** и **char** являются *примитивными*. Значением типа **array**(T_e, T_i) является *массив* с *элементами массива* типа T_e и *индексами* конечного типа T_i.

Гиперфункция – программа с несколькими *ветвями* результатов. Гиперфункция B(x**:** y**:** z) имеет две ветви результатов y и z. Исполнение гиперфункции завершается одной из ветвей с вычислением результатов по этой ветви; результаты других ветвей не вычисляются.

Вызов гиперфункции записывается в виде B(x**:** y #M1**:** z #M2). Здесь M1 и M2 – метки программы, содержащей вызов. Операторы перехода #M1 и #M2 встроены в ветви вызова. Исполнение вызова либо завершается первой ветвью с вычислением y и переходом на метку M1, либо второй ветвью с вычислением z и переходом на метку M2.

Вызов гиперфункции может комбинироваться с операторами обработки ветвей:

B(x**:** y #M1**:** z #M2) **case** M1: C(y**:** u) **case** M2: D(z**:** u) .

Вызов вида B(x**:** y #M1**:** z #M2); M1**: ...** может быть представлен оператором B(x**:** y**:** z #M2).

Формально гиперфункция определяется через предикатную программу следующего вида:

```
B(x: y, z, e)
pre P(x) post e = E(x) & (E(x) ⇒ S(x, y)) & (¬E(x) ⇒ R(x, z))
{ … };
```

Здесь x, y и z – непересекающиеся возможно пустые наборы переменных; P(x), E(x), S(x, y) и R(x, z) – логические утверждения. Предположим, что все присваивания вида e = **true** и e = **false** – последние исполняемые операторы в программе B. Программа B может быть заменена следующей программой в виде *гиперфункции*:

```
B(x: y #1:  z #2)
pre P(x)   pre 1: E(x) post 1: S(x, y) post 2: R(x, z)
{ … };
```

В теле гиперфункции каждое присваивание e = **true** заменено оператором перехода #1, а e = **false** – на #2. *Метки* 1 и 2 – дополнительные параметры, определяющие два различных *выхода* гиперфункции.

Спецификация гиперфункции состоит из двух частей. Утверждение после “**pre** 1” есть предусловие первой ветви; предусловие второй ветви – отрицание предусловия первой ветви. Утверждения после “**post** 1” и “**post** 2” есть постусловия для первой и второй ветвей, соответственно.

Аппарат *гиперфункций* является более общим и гибким по сравнению с известным механизмом обработки исключений, например, в таких языках, как Java и C++. Традиционные подходы в реализации обработки аварийных ситуаций предполагают заведение дополнительных структур, усложняющих программу. Этого удается избежать при использовании гиперфункций. Использование гиперфункций делает программу короче, быстрее и проще для понимания.

3. Трансформация программы перевода строки в целое число

3.1. Описание программы

Функция kstrtoull переводит целое число, представленное строкой, в значение типа unsigned long long, то есть неотрицательное (беззнаковое) целое максимального размера. Строка в языке Си – последовательность литер, завершающаяся нулевым байтом, то есть литерой '\0'. Строка – первый аргумент функции kstrtoull. Второй аргумент – основание числа в строке (8,10,16) или 0; в случае нуля основание числа должно быть определено по синтаксическому представлению числа во входной строке. Третий аргумент – указатель на переменную, в которую записывается значение числа. Возвращаемое значение функции kstrtoull – *код завершения*:

- 0 – нормальное завершение с вычисленным значением по третьему аргументу;
- -EINVAL – синтаксическая ошибка в представлении числа;
- -ERANGE – значение числа не помещается в тип unsigned long long.

Функция kstrtoull вызывает функцию _kstrtoull, которая далее вызывает функции _parse_integer и _parse_integer_fixup_radix. Во всех этих функциях первый параметр s – указатель на массив байтов (типа **char**), представляющий последовательность литер входной строки. Второй параметр base – основание числа: 8, 10 или 16, либо 0.

В соответствии с правилами языка Си в функции kstrtoull используется следующее представление целого числа в виде строки литер:

+ <основание числа> <последовательность цифр> '\n' '\0'

Здесь литера '\0' – нулевой байт, завершающий строку. Литера '\n' определяет переход на следующую строку. Знак числа «+», основание числа и литера '\n' могут отсутствовать в строковом представлении числа.

Ниже приведен код всех функций программы после приведения к нормальному виду условий в условных операторах и цикле **while**. Исходный код и его описание находятся на сайте: https://elixir.bootlin.com/linux/latest/source/lib/kstrtox.c.

Функция _parse_integer_fixup_radix вычисляет основание числа в случае, когда исходное значение параметра base равно нулю. В данной функции параметр base выступает как аргумент и результат. Поэтому он представлен указателем. Кроме того, для шестнадцатеричного числа в массиве s пропускается комбинация «0x». Результатом функции является продвинутый указатель по строке s, который должен указывать на первую цифру числа.

Отметим, что в соответствии с правилами языка Си шестнадцатеричное число начинается комбинацией литер «0x». Далее, восьмеричное число начинается с нулевой цифры. В остальных случаях число считается десятичным.

```
const char *_parse_integer_fixup_radix(const char *s, unsigned int *base)
{
        if (*base == 0) {
                if (s[0] == '0') {
                        if (_tolower(s[1]) == 'x' && isxdigit(s[2]))
                                *base = 16;
                        else
                                *base = 8;
                } else
                        *base = 10;
        }
        if (*base == 16 && s[0] == '0' && _tolower(s[1]) == 'x')
                s += 2;
        return s;
}
```

Функция _tolower преобразует заглавные буквы к нижнему регистру, сохраняя значения остальных литер. Функция isxdigit проверяет, является ли литера шестнадцатеричной цифрой.

Далее код функции _parse_integer. Она вычисляет собственно значение целого числа по последовательности цифр в строке по указателю s. Параметр base – основание числа: 8, 10 или 16. Вычисленное значение целого числа записывается в переменной по указателю p, третьему параметру функции. Возвращаемый результат функции – число цифр, составляющих исходное число и обработанных функцией. В случае переполнения, когда вычисляемое значение больше максимального значения ULLONG_MAX, в возвращаемый результат функции добавляется бит KSTRTOX_OVERFLOW (в 31 разряде), сигнализирующий о переполнении.

```
unsigned int _parse_integer(const char *s, unsigned int base, unsigned long long *p)
{
        unsigned long long res;
        unsigned int rv;
        res = 0;
        rv = 0;
        while (true) {
                unsigned int c = *s;
                unsigned int lc = c | 0x20; /* don't tolower() this line */
                unsigned int val;
                if ('0' <= c && c <= '9')
                        val = c - '0';
                else if ('a' <= lc && lc <= 'f')
                        val = lc - 'a' + 10;
                else
                        break;
                if (val >= base)
                        break;
                if (res & (~0ull << 60)) {
                        if (res > div_u64(ULLONG_MAX - val, base))
                                rv |= KSTRTOX_OVERFLOW;
                }
                res = res * base + val;
                rv++;
                s++;
        }
        *p = res;
        return rv;
}
```

Конструкция c | 0x20 эквивалентна вызову _tolower(c). Функция div_u64 реализует целочисленное деление. На каждом шаге цикла вычисления значения целого числа проводится контроль переполнения очередного значения за границы типа unsigned long long. Факт переполнения фиксируется битом KSTRTOX_OVERFLOW в результате функции. Однако само вычисление значения не прекращается, поскольку прерывание при этом не генерируется. Вычисление, конечно, будет неверным. Цикл продолжается до полного исчерпания набора цифр. Видимо, преследуется очевидная цель – отсканировать всю последовательность цифр не изменяя программу.

Ниже приведен код функции _kstrtoull, реализующей вычисление целого числа, представленного основанием числа и набором цифр. В ее теле вызываются две предыдущие функции _parse_integer_fixup_radix, вычисляющее основание, которое далее используется в вызове _parse_integer для получения значения числа. Дальнейшие действия реализуют контроль возможных ошибок в представлении числа. Вычисленное значение числа записывается в переменную по указателю res, третьему параметру функции. Возвращаемым результатом функции является код завершения: 0 – нормальное завершение без ошибок, -EINVAL – синтаксическая ошибка в представлении числа, -ERANGE – число слишком длинное и не помещается в тип unsigned long long.

```
static int _kstrtoull(const char *s, unsigned int base, unsigned long long *res)
{
	unsigned long long _res;
	unsigned int rv;
	s = _parse_integer_fixup_radix(s, &base);
	rv = _parse_integer(s, base, &_res);
	if (rv & KSTRTOX_OVERFLOW)
		return -ERANGE;
	if (rv == 0)
		return -EINVAL;
	s += rv;
	if (*s == '\n')
		s++;
	if (*s != '\0')
		return -EINVAL;
	*res = _res;
	return 0;
}
```

Далее код главной функции kstrtoull. Она вычисляет значение целого числа по его строковому представлению в параметре s. Второй параметр base – основание числа: 8, 10 или 16, либо 0, когда основание числа должно быть определено по синтаксическому представлению числа во входной строке. Третий параметр функции res – указатель на переменную, куда записывается вычисленное значение числа. Возвращаемым результатом функции является код завершения.

```
int kstrtoull(const char *s, unsigned int base, unsigned long long *res)
{
	if (s[0] == '+')
		s++;
	return _kstrtoull(s, base, res);
}
```

3.2. Устранение указателей

Для упрощения дедуктивной верификации программа kstrtoull трансформируется в эквивалентную предикатную программу. На первом этапе трансформаций устраняются указатели. Операции с указателями заменяются эквивалентными действиями без указателей. Устранение указателей существенно упрощает программу.

В программе kstrtoull используются указатели на строковые массивы и указатели на переменные. Для них применяются разные виды трансформаций. Трансформации могут быть применены при соблюдении условий, проверяемых с использованием специального потокового анализа программы.

3.2.1. Трансформация представления результатов функции

В языке Си всякая функция может иметь единственный результат, который передается оператором **return** в теле функции. Другой результат, если такой имеется, реализуется через параметр-указатель присваиванием переменной, на которую ссылается данный указатель. Одна из применяемых трансформаций – замена результатов функции *параметрами-результатами* в стиле языка предикатного программирования P [2]. Основной результат, передаваемый оператором **return**, и другие дополнительные результаты помещаются после разделителя «**:**» во второй секции параметров.

Для результата функции, передаваемого оператором **return**, вводится имя параметра. Например, для кода завершения вводится имя rep. Реализуется трансформация замены оператора **return** оператором присваивания параметру, например:

return 0 → rep = 0

Новая переменная вводится не всегда. Иногда используется локальная переменная, которая становится параметром-результатом, а оператор **return** устраняется из программы.

При наличии двух результатов функции присваивание результатам вызова функции представляется в виде оператора мульти-присваивания:

|res1, res2| = Имя функции(входные параметры)

3.2.2. Трансформации операций с указателями на переменные

Параметр-указатель, используемый для присваивания переменной, дополнительному результату функции, при трансформации заменятся параметром-результатом с прямым доступом, не через указатель. Для различных вхождений параметров-указателей в теле функции применяются следующие трансформации:

```
*p →  p
*base → base
*res →  res
&_res →  _res
&base → base
```

Если переменная, доступная по параметру-указателю, модифицируется в теле функции, то в трансформированной программе такая переменная будет находиться одновременно в составе аргументов и результатов.

3.2.3. Трансформации операций с указателями на массивы

Тип указателя **char** * заменяется типом массива:

char * → **array(char, nat)**

Вместо указателя s в трансформированной программе используется массив s и переменная js – индекс в массиве s. Значение индекса js в массиве s в точности соответствует позиции указателя s в исходной программе на языке Си.

Определим трансформации для различных вхождений s в исходной программе:

```
*s → s[js]
s++  → js++
s += 2 → js += 2
s[0] → s[js]
s[1] → s[js+1]
_kstrtoull(s, ...) → _kstrtoull(s, js, ...)
```

Первый параметр всех функций – строку s будет удобнее определить глобальной переменной в трансформированной программе. С учетом этого вызов функции преобразуется к виду _kstrtoull(js, ...).

3.2.4. Программа после устранения указателей

Для упрощения верификации полезно вынести константные параметры функций в состав глобальных переменных. Определим глобальный массив s.

array(char, nat) s;

При этом переменная js остается параметром. Ее тип – size_t, соответствующий адресуемой памяти.

В функции _parse_integer_fixup_radix параметр base используется и присваивается. Поэтому он помещается также среди результатов. Вхождения *base везде заменяются на base. Оператор **return** s устраняется, а образ s, т.е. массив s и индекс js, надо было бы поместить среди результатов функции. Однако массив s – глобальный, поэтому в результаты помещается только js.

```
_parse_integer_fixup_radix(size_t js,  unsigned int base : unsigned int base, size_t js)
{
        if (base == 0) {
                if (s[js] == '0') {
                        if (_tolower(s[js+1]) == 'x' && isxdigit(s[js+2]))
                                base = 16;
                        else
                                base = 8;
                } else
                        base = 10;
        }
        if (base == 16 && s[js] == '0' && _tolower(s[js+1]) == 'x')
                js += 2;
}

_parse_integer(size_t js, unsigned int base : unsigned long long p, unsigned int rv)
{
        unsigned long long res;
        res = 0;  rv = 0;
        while (true) {
                unsigned int c = s[js];
                unsigned int lc = c | 0x20; /* don't tolower() this line */
                unsigned int val;
                if ('0' <= c && c <= '9')
                        val = c - '0';
                else if ('a' <= lc && lc <= 'f')
                        val = lc - 'a' + 10;
                else
                        break;
                if (val >= base)
                        break;
                if (res & (~0ull << 60)) {
                        if (res > div_u64(ULLONG_MAX - val, base))
                                rv |= KSTRTOX_OVERFLOW;
                }
                res = res * base + val;
                rv++;
                js++;
        }
        p = res;
}
```

Выше представлена функция _parse_integer после трансформации. Переменная p становится результатом функции. Оператор *p = res заменяется на p = res. Оператор **return** rv устраняется. Локальная переменная rv становится параметром-результатом функции.

Ниже код функции _kstrtoull после трансформации. Переменная res становится результатом функции. Оператор *res = _res заменяется на res = _res. Вводится переменная-результат rep для кода завершения, возвращаемого оператором **return**. Оператор **return** 0 заменяется присваиванием rep = 0. Оператор **return** -ERANGE заменяется фрагментом {rep = -ERANGE; **return**}. Поскольку функции _parse_integer_fixup_radix и _parse_integer имеют два результата, присваивание результатов вызовов реализуется оператором мульти-присваивания.

```
_kstrtoull(size_t js, unsigned int base : unsigned long long res,  int rep)
{
        unsigned long long _res;
        unsigned int rv;
        |base, js| = _parse_integer_fixup_radix(js, base);
        |_res, rv | = _parse_integer(js, base);
        if (rv & KSTRTOX_OVERFLOW)
                {rep = -ERANGE; return}
        if (rv == 0)
                {rep = -EINVAL; return}
        js += rv;
        if (s[js] == '\n')
                js++;
        if (s[js] != zero)
                {rep = -EINVAL; return}
        res = _res;
        rep = 0;
}
```

Трансформация главной функции kstrtoull вводит переменные-результаты res и rep. Параметр-указатель s заменяется глобальным массивом s и локальной переменной – индексом js с начальной инициацией size_t js =0. Вхождение s[0] заменяется на s[js+0], равное s[0].

```
kstrtoull(unsigned int base : unsigned long long res, int rep)
{       size_t js =0;
        if (s[0] == '+')
                js++;
        |res, rep| = _kstrtoull(js, base);
}
```

Распознавание того, что параметр-указатель s подставляется аргументом в вызове _kstrtoull, а res подставляется результатом, реализуется потоковым анализом тела функции _kstrtoull.

3.3. Трансформации в предикатную программу

Программа после устранения указателей переводится в предикатную программу. Циклы заменяются рекурсивными программами. Некоторые фрагменты программы оформляются гиперфункциями.

Преобразуем имена типов:

```
type = nat32 = 0..2**32 – 1; // unsigned int;
type = nat64 = 0..2**64 – 1; // unsigned long long;
type = size_t = nat64;
```

Определим глобальный массив s.

```
array(char, nat) s;
```

Из программы _parse_integer вынесем фрагмент вычисления значения очередной цифры.

```
digitVal(nat32 c : nat32 val : #notDigit){
            nat32 lc = c | 0x20; /* don't tolower() this line */
            if ('0' <= c && c <= '9')
                  val = c - '0';
            else if ('a' <= lc && lc <= 'f')
                  val = lc - 'a' + 10;
            else
                  #notDigit
            if (val >= base)
                  #notDigit
}
```

Первая ветвь гиперфункции digitVal соответствует нормальному завершению программы с вычислением значения val для цифры c. Выход #notDigit по второй ветви гиперфункции соответствует случаю, когда литера c – не цифра.

В программе _parse_integer заменим цикл **while** рекурсивной программой. Ее дополнительными аргументами становятся переменные js, res и rv, модифицируемые в цикле.

```
parseInt (size_t js, nat64 res, nat32 rv, base : nat64 res, nat32 rv)
{
            digitVal(s[js] : nat32 val : return);
            if (res & (~0ull << 60)) {
                  if (res > div_u64(ULLONG_MAX - val, base))
                        rv |= KSTRTOX_OVERFLOW;
            }
            parseInt (js+1, res * base + val, rv+1, base :  res, rv)
}
```

Отметим, что тип переменной js определен как size_t, соразмерно памяти, доступной по исходному указателю s.

Программа _parse_integer преобразуется к виду:

```
_parse_integer(size_t js, nat32 base : nat64 p, nat32 rv)
{
        parseInt(js, 0, 0, base: nat64 res, rv);
        p = res;
}
```

Программа _parse_integer_fixup_radix остается без изменений.

```
_parse_integer_fixup_radix(size_t js, nat32 base : nat32 base, size_t js)
{
        if (base == 0) {
                if (s[js] == '0') {
                        if (_tolower(s[js+1]) == 'x' && isxdigit(s[js+2]))
                                base = 16;
                        else
                                base = 8;
                } else
                        base = 10;
        }
        if (base == 16 && s[js] == '0' && _tolower(s[js+1]) == 'x')
                js += 2;
}
```

Программы kstrtoull и _kstrtoull заменяются гиперфункциями с тремя ветвями. Первая ветвь гиперфункции соответствует нормальному завершению с вычислением значения res для целого числа. Вторая ветвь завершается выходом #ERange в случае выхода значения константы за пределы типа nat64. Третья ветвь завершается выходом #ESint при обнаружении синтаксической ошибки в представлении числа. Устраняется переменная-результат – код завершения rep.

```
_kstrtoull(size_t js, nat32 base : nat64 res #Valid : #ERange : #ESint)
{
        nat64 _res;
        nat32 rv;
        |base, js| = _parse_integer_fixup_radix(js, base);
        |_res, rv | = _parse_integer(js, base);
        if (rv & KSTRTOX_OVERFLOW)
                #ERange
        if (rv == 0)
                #ESint
        js += rv;
        if (s[js] == '\n')
                js++;
        if (s[js] != zero)
                #ESint
        res = _res;
        #Valid
}

kstrtoull(nat32 base : nat64 res #Valid : #ERange : #ESint)
{       size_t js =0;
        if (s[js] == '+')
                js++;
        _kstrtoull(js, base: res #Valid: #ERange : #ESint);
}
```

4. Спецификация предикатной программы

Полная предикатная программа представлена выше набором функций. Спецификация программы содержит предусловие и постусловие для каждой функции.

Для упрощения верификации из предикатной программы в максимальной степени экстрагирована ее константная часть, отражающая внутренние состояния исполняемой программы. При вынесении константной части предикатная программа подверглась существенной модификации. Большая часть аргументов программ стали глобальными переменными. Экстрагированная часть определяет модель программы, см. Рис. 1. Отметим, что в предыдущий релиз дедуктивной верификации базировался на обычном стиле спецификации программы без модели. Доказательство формул корректности в системе Why3[18] оказалось громоздким и трудоемким.

Последующие изменения, реализованные при доказательстве формул корректности, в спецификации и генерации формул корректности отмечены другими цветами. Разные цвета соответствуют разным сериям изменений.

4.1. Базисная часть

Модель программы определяется набором переменных js0, ks, n, sL, resN и аксиом, определяющих их свойства. Переменная js0 определяет позицию после возможного "+" в начале строкового представления числа; ks определяет позицию первой цифры после возможного радекса, задающего основание числа (8,10,16); n фиксирует позицию после последней цифры; sL определяет позицию последней нулевой литеры '\0'; resN – значение числа.

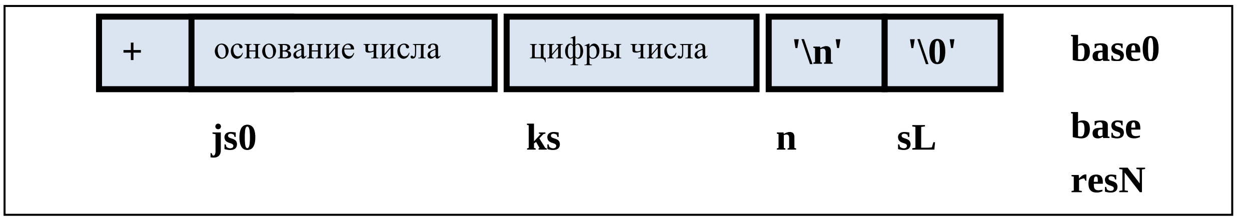

Рис. 1. Модель программы

Будем считать, что в типе **char** нет верхнего регистра для букв, что достигается применением функции _tolower. Как следствие, не рассматривается литера 'X' и опускаются вызовы функции tolower в программе. Введем константы:

```
char zero = '\0', nul = '0', nl = "\n", iks = "x", plu= "+";
```

Параметр base определяет систему счисления.

```
formula isBase(nat base) = base = 8 or base = 10 or base = 16;
type arCh = array(char, nat);
arCh s; (*массив литер s – соответствует входной строке функции kstrtoull *)
nat  sL; (* индекс последней литеры исходной строки s*)
axiom ALnat: sL >= 0;
axiom ALs: s[sL] = zero;
nat base0; (*начальное значение второго аргумента kstrtoull *)
axiom Ab0: base0 = 0 or isBase(base0);
```

Индекс после плюса:

```
nat js0 = if s[0] = plu then 1 else 0;
formula radix(nat j, k, base) =
          (if base0=0 then
             (if s[j] = '0' then (if s[j+1]='x' then base = 16  else  base =8)
              else base =10 )
           else base = base0)
          &
          (if base=16 & s[j] = '0' & s[j+1='x' then k=j+2 else k=j);
nat  ks; (* индекс начальной цифры в исходной строке s *)
nat base; (*итоговое *)
axiom Kse: radix(js0, ks, base);
```

Таким образом, base0 определяет начальное значение исходного параметра, а base – его значение после вызова _parse_integer_fixup_radix.

Цифры как подмножество типа **char** определяются отношением digit:

```
formula digitB(char c, nat base, valD);
axiom Adig:  forall char c, nat base, valD. isBase(base) ⇒
          ( 220<=c<220+base <->  digitB(c, base, valD) /\ c = valD + 220)
```

Нет необходимости определять детально, как по цифре получается ее значение. Достаточно использовать данную аксиому. Здесь предполагается, что шестнадцатеричные цифры следуют непосредственно за десятичными, что в действительности не так. Это упрощение полезно для доказательства формул корректности.

```
formula  digit(char c, nat valD) = digitB(c, base, valD);
formula isDigit(char c) = ∃nat valD. digit(c, valD);
formula isDigit16(char c) = ∃nat valD. digitB(c, 16, valD);
```

Формула digit определяет цифру в системе счисления, определяемой глобальной переменной base.

Формула endNum определяет индекс литеры после последовательности цифр.

```
formula endNum(nat n) = n >= ks & ¬isDigit(s[n]) & ∀j=ks..n-1. isDigit(s[j]);
nat  n; (* индекс за последней цифрой в строке s*)
axiom Anu: endNum(n);

formula numRes(nat k, m, nat res0, res) =
          if (k=m) res = res0
          else ∃val. digit(s[k], val) & numRes(k+1, res0 * base + val, res);
```

Формула numRes вычисляет значение числа res по оставшейся последовательности цифр в вырезке s[k .. m] в предположении, что res0 – значение числа по предыдущей последовательности цифр. Отметим, что данное рекурсивное определение не воспринимается в Why3. Вместо него пришлось использовать индуктивное определение, которое затем дважды подвергалось модификации.

```
formula number(nat res) = numRes(ks, n, 0, res);
nat resN;
axiom Ares: number(resN);
formula afterNum() = ks<n & (s[n]= zero \/ s[n]=nl /\ s[n+1]= zero);
```

Формула afterNum постулирует, что по индексу n находится либо нулевой байт, либо перевод строки с последующим нулевым байтом. Условие ks<n здесь необходимо. Оно добавлено позже при дедуктивной верификации.

Определенная выше модель программы фактически представляет специализацию входных аргументов base и js с фиксацией свойств каждой специализации, что позволяет кардинально упростить доказательство формул корректности.

4.2. Спецификация программы kstrtoull

Программа kstrtoull модифицирована в соответствии с моделью. Аргументы перенесены в глобальные переменные. Аргументы исчезли также и в вызове _kstrtoull. Вхождения переменной js заменены на js0.

```
kstrtoull( : nat64 res #Valid : #ERange : #ESint)
pre Valid: pValid() pre ERange: pERange() pre ESint: pESint()
post Valid: qValid(res)
{       ~~size_t js;~~
        if (s[0] == '+') js0 = 1 else js0 =0;
        _kstrtoull(~~js, base~~ : res #Valid : #ERange : #ESint);
}
```

При отсутствии аргументов нет общего предусловия. Но есть предусловия по ветвям гиперфункции. Предусловие третье ветви ESint далее определяется как дополнение предусловий первых двух ветвей в рамках общего предусловия.

```
nat max64 = 2**64 – 1; // = ULLONG_MAX
formula pValid() = isDigit(s[ks]) & afterNum() & resN <= max64;
formula pERange() = isDigit(s[ks]) & resN > max64;
lemma Lpre12:  ¬ (pValid() & pERange())
formula pESint() = not pValid() /\ not pERange();
lemma Lpre3: not (pValid() /\ pESint());
formula qValid(nat64 res) =  isDigit(s[ks]) & res = resN
formula qkstrtoull(nat64  res, nat e) =
    (e = 0 -> pValid() /\ qValid(res)) /\
    (e = 1 -> pERange()) /\
    (e = 2 -> pESint());
```

При дедуктивной верификации гиперфункция kstrtoull преобразуется в функцию с дополнительным результатом e. Постусловие qkstrtoull этой функции составляется указанным образом из постусловий и предусловий ветвей.

4.3. Спецификация программы _kstrtoull

```
formula q_kstr(nat64 res, nat c) =
    (c = 0 -> pValid() /\ qValid(res)) /\
    (c = 1 -> pERange()) /\
    (c = 2 -> pESint())
```

Формула q_kstr определяет общее постусловие для гиперфункции _kstrtoull. Предусловия и постусловия ветвей наследуются от главной программы-гиперфункции.

```
_kstrtoull( : nat64 res #Valid : #ERange : #ESint)
pre Valid: pValid() pre ERange: pERange() pre ESint: pESint()
post Valid: qValid(res)
{
        _parse_integer_fixup_radix( : base, ks);
        _parse_integer( : nat64 _res, nat32 rv, bool of);
        if (of) #ERange
        else if (rv == 0) #ESint
        else { size_t js2 = ks + rv;
                if (s[js2] == '\n') js3 = js2+1 else js3 = js2;
                if (s[js3] != zero) #ESint
                else {res = _res; #Valid }
        }
}
```

Переменная of истинна в случае переполнения при вычислении значения числа.

4.4. Спецификация программы _parse_integer_fixup_radix

```
_parse_integer_fixup_radix( : nat base, size_t js)
post radix(js0, js, base)
{
        if (base0 == 0) {
                if (s[js0] == '0') {
                        if (s[js0+1]) == 'x' (*& isxdigit(s[js0+2])*)) base = 16
                        else base = 8;
                } else  base = 10;
        };
        if (base == 16 & s[js0] == '0' & s[js0+1] == 'x') js= js0 + 2 else js = js0;
}
```

Вызов isxdigit(s[js0+2]) является избыточным и он удален. Если не шестнадцатеричная цифра, то base = 8 и далее диагностируется синтаксическая ошибка. Функция _parse_integer_fixup_radix находится в интерфейсе пользователя библиотекой ОС Linux. Однако подобная особенность нигде не оговаривается. Поэтому вызов sxdigit(s[js0+2]) не является необходимым и для независимого вызова _parse_integer_fixup_radix.

```
formula isxdigit(char c: bool) = isDigit(c, 16);
```

4.5. Спецификация программы _parse_integer

В данной программе три ошибки. Хотя внесены они вполне сознательно с расчетом: крайне маловероятно, что в реальных приложениях кто-то от них не пострадает. Тип переменная rv выше определен как nat32. При достаточно длинной последовательности цифр значение rv выйдет за границу типа nat32. Оператор rv |= KSTRTOX_OVERFLOW исходной программы также вставлен с расчетом недостижимости 32-го бита значением rv.

При дедуктивной верификации эти ошибки неизбежно фиксируются. Чтобы эти ошибки не мешали проведению верификации, вместо бита KSTRTOX_OVERFLOW введена переменная of, регистрирующая переполнение итогового значения числа. Тип переменной rv определен как **nat**. Переполнение при вычислении целого числа хотя и фиксируется, однако далее вычисление, приводящее к переполнению, все же зачем-то производится. Видимо, в предположении, что оно не приведет к аварийному завершению исполнения. Возможно для того, чтобы отсканировать число до конца. Однако это нигде не оговорено. Чтобы купировать данную ошибку при верификации, для переменной res вместо nat64 используется тип **nat**.

```
formula qParse(nat64 p, nat rv, bool of) =
                    rv = n – ks & of = (p>max64) & p = resN;

_parse_integer(  : nat64 p, nat rv, bool of)
post qParse(p, rv, of)
{       parseInt(ks, 0, 0, false: nat64 res, rv, of);
        p = res;
}
```

Проведена замена res & (~0ull << 60) на res >= 2**60. При строгом подходе эквивалентность такой замены необходимо формально доказывать. Занятие это утомительное и неблагодарное. Вызов digitVal заменен на композицию **if** (isDigit(s[js], 16)) digit(s[js], 16, val).

```
formula pParseInt(size_t js, nat64 res, nat rv, bool of0) =
               ks<=js<=n & (∀nat j=ks..js-1. isDigit(s[j])) &
                              numRes(ks, js, 0, res) & rv = js-ks & of0 =(res>max64);
formula qParseInt(size_t js, nat64 res, res’, nat rv, rv’, of) =
     numRes(js, n, res, res’) & rv’ = rv + n – js & of = res’>max64;

parseInt (size_t js, nat64 res, nat rv, bool of0: nat res’, nat rv’, bool of)
pre pParseInt(js, res, rv, of0) post qParseInt(js, res, res’, rv, rv’, of) measure sL – js;
{       if (isDigit(s[js])) {
                digit(s[js], val);
~~              nat res1 = res*base + val;~~
                if (res >= 2**60 & res > (max64 – val) / base) of= true else of = of0;
                parseInt (js+1, res*base + val, rv+1, of0 :  res’, rv’, of);
        } else { res’ = res || rv’ = rv || of = of0 }
}
```

5. Процесс дедуктивной верификации

По завершению построения спецификации была проведена генерация формул корректности применением системы правил [7]. Построение формул корректности подробно документировано в Приложении 3. Формулы корректности вместе со спецификацией собраны в теории, представленные в Приложении 1. Теории были переведены на язык спецификаций системы Why3 [18].

Верификация в системе Why3 оказалась тяжелой. Естественный и привычный стиль спецификации иногда приводит к громоздким доказательствам формул корректности, когда доказательство любого простого утверждения требует длительной работы. Чтобы упростить верификацию, из предикатной программы в максимальной степени экстрагирована ее константная часть в виде модели, описанной в разд. 4.1, Рис. 1. Фактически проведена специализация по аргументам base и js, что позволило упростить программу и спецификацию.

В процессе верификации последовательно исправлялись ошибки в спецификации. Трижды модифицировалась программа parseInt. Применялись разные способы обхода ошибок, указанных в разд. 4.5. Конечные теории по завершению процесса верификации представлены в Приложении 2.

Две леммы NuEq и NuNext, требующие доказательства по индукции, доказаны с помощью аппарата лемма-функций [17]. Для этого строится вспомогательная предикатная программа, спецификация которой совпадает с исходной леммой. Покажем эту ставшую популярной технику на примере леммы NuEq.

```
lemma NuEq :   forall k:int, m:int, res0:int, res:int, res1:int.
          numRes k m res0 res /\ numRes k m res0 res1 -> res1 = res
```

Строится следующая предикатная программа.

```
NuEq(int k, m, res, res0, res1)
   post numRes(k, m, res0, res) & numRes(k, m, res0, res1) ⇒ res1 = res;
   measure m
{   if (k = m) {  }  else  { NuEq(k, m-1, res, res0, res1)} }
```

Программа не обязана быть правильной. Поэтому первая ветвь условного оператора пустая. Постусловие здесь в точности формула леммы NuEq. По данной программе генерируется следующая формула корректности:

RB5: k <> m & qNuEq(k, m-1, res, res0, res1) ⇒ qNuEq(k, m, res, res0, res1)

Используя формулу **RB5**, SMT-решатели Z3 и CVC4, входящие в составе системы Why3 [18], уже способны автоматически доказать лемму NuEq.

Соответствующие программы и генерация формул корректности для лемма-функций приведены в конце Приложения 3. Были трудности с построением правильных программ для лемма-функций. При доказательстве последней леммы NuNext обнаружилась ошибка в индуктивном определении предиката numRes: индуктивно порождаемое множество оказалось пустым.

Дополнительных лемм относительно немного: 4 леммы при доказательстве формул корректности программы parseInt и 15 лемм как расширение исходной модели. В системе Coq проведено только три коротких доказательства.

6. Обзор работ

Трансформации, устраняющие указатели – новое направление исследований. Наиболее близкими здесь являются работы по анализу видов структур данных (shape analysis) [10, 12, 13] и обратной трансляции [14,16] на язык более высокого уровня. Причем в этих работах не ставится задача устранения указателей.

Наборы трансформаций для устранения указателей, применяемых для разных программ на языке Си, отличаются. И пока рано говорить о создании универсальной системы трансформаций. Трансформации для программы конкатенации строк [6] оперируют с несколькими указателями на одном массиве, чего нет в настоящей работе. Трансформации устранения указателей для программы пирамидальной сортировки [5] более просты: там не требуется введения переменных-индексов. В настоящей работе используется больше видов трансформаций. Новыми являются трансформации для параметров-указателей и вхождений указателей в качестве фактических параметров вызовов функций.

Параметр-указатель оказывается необходимым для второго результата функции, поскольку в языке Си всякая функция может иметь единственный результат, передаваемый оператором **return** в теле функции. В языке предикатного программирования P [2] и языке функционального программирования WhyML [18] допускается несколько результатов. Кроме того, в языке P допускается несколько выходов в гиперфункциях, причем каждый выход может иметь несколько результатов. Схожие по выразительности возможности в языке WhyML: исключения с несколькими параметрами.

Вычисление значения целого числа по его строковому представлению представляется в виде стандартной библиотечной функции практически во всех популярных языках программирования. Тем не менее, формальной верификации этой функции, по-видимому, никогда не проводилось.

7. Итоги верификации

Казалось бы, программа kstrtoull проста, и в ней не должно быть ошибок. Однако иногда ошибки вносятся сознательно в целях оптимизации программы по времени исполнению или по размеру объектного кода. Здесь расчет на то, что в реальных приложениях такие ошибки никогда не проявятся. Разумеется, такая практика провоцирует дурной стиль программирования.

Программа _parse_integer содержит следующие ошибки:

1. Тип переменная rv определен как **unsigned int**. При достаточно длинной последовательности нулевых цифр значение rv выйдет за границу типа. Для устранения ошибки необходим контроль выхода за границы типа с выдачей дополнительного флага ошибки. Более правильно было бы определить тип rv как **size_t**, соразмерно типу указателя переменной s.

2. При достаточно длинной последовательности цифр значение переменной rv будет содержать единицу в 32-м бите, что будет диагностировано как переполнение. Поэтому ошибочно использовать оператор rv |= KSTRTOX_OVERFLOW для диагностики переполнения.

3. Контроль переполнения при вычислении целого числа реализуется. Однако вычисление, приводящее к переполнению, все же зачем-то производится. Что, безусловно, ошибочно.

Данные ошибки хорошо известны. Они не исправляются по той причине, что на реальных приложениях эти ошибки не проявились.

4. В верифицируемой программе _parse_integer_fixup_radix удалена проверка условия isxdigit(s[2]). В результате такого изменения, если после «0x» далее следует не шестнадцатеричная цифра, то программа выдаст base=16, хотя ранее выдавала base=8. Для ошибочного числа это не принципиально. Поэтому следует рекомендовать устранить проверку isxdigit(s[2]) в библиотеке стандартных программ ОС Linux.

В остальном, программа kstrtoull в точности соответствует спецификации и не содержит других ошибок.

Список литературы

1. Доказательство правил корректности операторов предикатной программы. [Электронный ресурс]. URL:http://www.iis.nsk.su/persons/vshel/files/rules.zip . (дата обращения 12.08.2020)
2. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования P. Версия 0.14. Новосибирск, 2018. 28с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/plang14.pdf. (дата обращения 12.12.2020)
3. Чушкин М.С. Система дедуктивной верификации предикатных программ. *Программная инженерия*. 2016. № 5. С. 202-210. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/paper.pdf. (дата обращения 12.12.2020)
4. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования. *Программная инженерия*, 2011, № 2. С. 14-21.
5. Шелехов В.И. Верификация предикатной программы пирамидальной сортировки с применением обратной трансформации. — ИСИ СО РАН, Новосибирск, 2020. 36с. [Электронный ресурс]. URL: https://persons.iis.nsk.su/files/persons/pages/sort9.pdf (дата обращения 12.12.2020)
6. Шелехов В.И. Дедуктивная верификация программы конкатенации строк с применением обратной трансформации // *Знания-Онтологии-Теории (ЗОНТ-19*). — Новосибирск, 2019. — 19с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/logcflc1.pdf. (дата обращения 12.08.2020)
7. Шелехов В.И. Правила доказательства корректности предикатных программ. — Новосибирск, ИСИ СО РАН, 2019. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/prrules.pdf (дата обращения 12.12.2020)
8. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. Новосибирск, 2012. 30с. (Препр. / ИСИ СО РАН. № 164).
9. Шелехов В.И., Чушкин М.С. Верификация программы быстрой сортировки с двумя опорными элементами. *Научный сервис в сети Интернет*. М.: ИПМ им. М.В.Келдыша, 2018. 26с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/dqsort.pdf. (дата обращения 12.08.2020)
10. Boockmann J.H., Lüttgen G., Mühlberg J.T. Generating Inductive Shape Predicates for Runtime Checking and Formal Verification // Leveraging Applications of Formal Methods, Verification and Validation. Verification. 2018. LNCS 11245. P. 64-74.
11. The Coq Proof Assistant. [Электронный ресурс]. URL: https://coq.inria.fr/. (дата обращения 12.08.2020)
12. Haller I., Slowinska A., Bos H. Scalable data structure detection and classification for C/C++ binaries // Empirical Software Engineering. 2016, v. 21, Issue 3. P. 778–810.
13. Jung C., Clark N. DDT: Design and evaluation of a dynamic program analysis for optimizing data structure usage // 42nd Int. Symposium on Microarchitecture (MICRO 42), NY, 2009. P. 56-66.
14. Novosoft C2J: a C to Java translator. http://www.novosoft-us.com/ solutions/product c2j.shtml, 2001.
15. Shelekhov V. I. Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements // Automatic Control and Computer Sciences. 2011. Vol. 45, No. 7, P. 421–427.
16. Trudel M., Furia C. A., Nordio M., Meyer B., Oriol M. C to O-O Translation: Beyond the Easy Stuff // 19th Working Conference on Reverse Engineering, 2012. P. 19-28.
17. Volkov, G., Mandrykin, M., Efremov, D.: Lemma functions for Frama-C: C programs as proofs. In: Proceedings of the 2018 Ivannikov ISPRAS Open Conference (ISPRAS-2018). pp. 31–38.
18. Why 3. Where Programs Meet Provers. [Электронный ресурс]. URL:http://why3.lri.fr. (дата обращения 12.08.2020)

Приложение 1

Теории для формул корректности

Определим набор теорий с формулами корректности для всех программ, а также набор формул, вызываемых в формулах корректности, в частности, в предусловиях и постусловиях.

```
theory Base {
char zero = '\0', nul = '0', nl = "\n", iks = "x", plu= "+";
formula isBase(nat base) = base = 8 or base = 10 or base = 16;
type arCh = array(char, nat);
arCh s; (*первый аргумент kstrtoull *)
nat  sL; (* индекс последней литеры исходной строки s*)
axiom ALnat: sL >= 0;
axiom ALs: s[sL] = zero;
nat base0; (*второй аргумент kstrtoull *)
axiom Ab0: base0 = 0 or isBase(base0);
nat js0 = if s[0] = plu then 1 else 0;
formula radix(nat j, k, base) =
           (if base0=0 then
              (if s[j] = '0' then (if s[j+1]='x' then base = 16  else  base =8)
               else base =10 )
            else base = base0)
           &
           (if base=16 & s[j] = '0' & s[j+1='x' then k=j+2 else k=j);
nat  ks; (* индекс начальной цифры в исходной строке s*)
nat base; (*итоговое *)
axiom Kse: radix(js0, ks, base);
formula digitB(char c, nat base, valD);
axiom Adig:  forall char c, nat base, valD. isBase(base) ⇒
            ( 220<=c<220+base <->  digitB(c, base, valD) /\ c = valD + 220)
formula  digit(char c, nat valD) = digitB(c, base, valD);
formula isDigit(char c) = ∃nat valD. digit(c, valD);
formula isDigit16(char c) = ∃nat valD. digitB(c, 16, valD);
formula endNum(nat n) = n >= ks & ¬isDigit(s[n]) & ∀j=ks..n-1. isDigit(s[j]);
nat  n; (* индекс за последней цифрой в строке s*)
axiom Anu: endNum(n);
formula numRes(nat k, m, nat res0, res) =
           if (k= m) res = res0
           else ∃val. digit(s[k], val) & numRes(k+1, res0 * base + val, res);
formula number(nat res) = numRes(k, n, 0, res);
nat resN;
axiom Ares: number(resN);
formula afterNum() = s[n]= zero \/ s[n]=nl /\ s[n+1]= zero;
} Base;
```

Формулы корректности программы **kstrtoull**

```
theory Kstrtoull {
import Base;
formula pValid() = isDigit(s[ks]) & afterNum() & resN <= max64;
formula pERange() = isDigit(s[ks]) & resN > max64;
lemma Lpre12:  ¬ (pValid() & pERange())
formula pESint() = not pValid() /\ not pERange();
lemma Lpre3: not (pValid() /\ pESint());
formula qValid(nat64 res) =  isDigit(s[ks]) & res = resN
formula qkstrtoull(nat64 res, nat e) =
        (e = 0 ⇒ pValid() & qValid(res)) &
        (e = 1 ⇒ pERange()) &
        (e = 2 ⇒ pESint())
formula S(size_t js) = if s[0] = '+' then js =1 else js =0;
formula q_kstr(nat64 res, nat c) =
      (c = 0 -> pValid() /\ qValid(res)) /\
      (c = 1 -> pERange()) /\
      (c = 2 -> pESint())
~~HSB2:  S(js) ⇒ p_kstr(js);~~
COR1: S(js0) & qValid( res) & pValid() & e=0 ⇒  qkstrtoull( res, e);
COR2: S(js0) & pERange() & e=1 ⇒ qkstrtoull( res, e);
COR3: S(js0) & pESint() & e=2 ⇒ qkstrtoull(res, e);
}Kstrtoull;
```

Формулы корректности программы _kstrtoull

```
theory Kstrtoull_ {
import Base, Kstrtoull;
type size_t = nat;
formula qParse(nat64 p, nat32 rv, bool of) =
                    rv = n – ks & of = (p>max64) & p = resN;
~~QSB1: p_kstr(js) & radix(js, js1, base1) ⇒ pParse(js1, base1)~~
formula p3(nat64 _res, nat32 rv, bool of) = radix(js0, ks, base) & qParse( _res, rv, of);

COR4: p3( _res, rv, of) & of & c=1 ⇒ q_kstr(res, c);
COR5: p3( _res, rv, of) & ¬of & rv = 0 & c=2 ⇒ q_kstr(res, c);
formula p4(size_t js2, nat64 _res, nat32 rv, bool of) =
      p3( _res, rv, of) & ¬of & ¬rv = 0 & js2 = ks + rv;
formula S3(size_t js2, js3) = if (s[js2] = '\n') js3 = js2+1 else js3 = js2;

COR6: p4(js2, _res, rv, of) & S3(js2, js3) & s[js3] != zero & c=2 ⇒ q_kstr(res, c);
COR7: p4(js2, _res, rv, of) & S3(js2, js3) & s[js3] = zero & res = _res & c=0  ⇒ q_kstr(res, c);
}Kstrtoull_;
```

Формулы корректности программы _parse_integer_fixup_radix

```
theory Parse_integer_fixup_radix {
import Base, Kstrtoull;
formula S4( nat base) =
      if base0 = 0 then  ( if s[js0] = '0' then
                                  ( if s[js0+1]) = 'x' then base = 16 else base = 8)
                         else  base = 10 )
      else base = base0;
```

QS1: ∃ base. S4(base);
formula C2(**nat** base) = base = 16 & s[js0] = '0' & s[js0+1] = 'x';
COR8: S4(base) & C2(base) & js= js0 + 2 ⇒ radix(js0, js, base);
COR9: S4(base) & ¬C2(base) & js = js0 ⇒ radix(js0, js, base);
} Parse_integer_fixup_radix;

Формулы корректности программы _parse_integer

theory Parse_integer {
import Base, Kstrtoull, Kstrtoul_;
formula pParseInt(size_t js, nat64 res, nat32 rv, **bool** of0) =
        ks<=js<=n & (∀**nat** j=ks..js-1. isDigit(s[j])) &
                numRes(ks, js, 0, res) & rv = js-ks & of0 =(res>max64);
formula qParseInt(size_t js, nat64 res, res′, nat32 rv, rv′, of) =
    numRes(js, n, res, res′) & rv′ = rv + n – js & of = res′>max64;
QSB2: pParseInt(ks, 0, 0, **false**);
formula qParse(nat64 p, nat32 rv, **bool** of) =
            rv = n – ks & of = (p>max64) & p = resN;
COR10: qParseInt(ks, 0, res, 0, rv, of) & p = res ⇒ qParse(p, rv, of);
formula m(size_t js) = sL – js;
RP1: pParseInt(js, res, rv, of0) & ¬isDigit(s[js]) & res′ = res & rv′ = rv & of = of0 ⇒
                qParseInt(js, res, res′, rv, rv′, of);
formula E6(**nat** res, val, **bool** of0, of1) =
        if (res >= 2**60 & res > (max64 – val) / base) **then** of1= **true** **else** of1 = of0
~~**RP3**: pParseInt(js, res, rv) & isDigit(s[js]) & digit(s[js], val) & res1 = res*base + val &~~
            ~~E6(res,val) & res′ = res1 & rv′ = rv & of= **true** & jl=js ⇒~~
                            ~~qParseInt(js , jl, res, res′, rv, rv′, of);~~
formula p6(size_t js, **nat** res, val, nat32 rv, **bool** of0, of1) =
    pParseInt(js, res, rv, of0) & isDigit(s[js]) & digit(s[js], val) & E6(res, val, of0, of1);
formula pB(size_t js, **nat** rv) = rv+1 < 2**32;
RB1: p6(js, res, val, rv, of0, of1) ⇒ pParseInt(js+1, res*base + val, rv+1, of1) & m(js+1)<m(js);
RB2: p6(js, res, val, rv, of0, of1) & qParseInt(js+1, res*base + val, res′, rv1+1, rv′, of) ⇒
        qParseInt(js, res, res′, rv, rv′, of)
}Parse_integer;

Приложение 2

Теории для формул корректности на языке why3

Теории, приведенные в Приложении 2, преобразованы на язык спецификаций системы верификации Why3. Здесь приведена финальная версия после проведнния всех доказательств.

```
theory Base
  use export int.Int
  use export map.Map

  type nat32 = int
  type nat64 = int
  type size_t = int
  type char = int
  constant zerO: char = 0
  constant nul: char = 220
  constant nl: char = 1
  constant iks: char = 2
  constant plu: char = 3

  type arCh = int -> char
  constant s: arCh                (*first parameter of kstrtoull*)
  constant sL: int
  axiom Alnat: sL >= 0
  axiom ALs: s[sL] = zerO

  predicate isBase(base: int) = base = 8 \/ base = 10 \/ base = 16
  constant base0: int           (*second kstrtoull  parameter *)
  axiom Ab0: base0 = 0 \/ isBase(base0)
  constant js0: int = if s[0] = plu then 1 else 0
  lemma Ljs0: js0 <= sL

  predicate radix(j k base: int) =
    (if base0=0 then
        (if s[j] = zerO then (if s[j+1]=iks then base = 16  else  base = 8)
         else base =10 )
     else base = base0)
    /\
    (if base=16 /\ s[j] = nul /\ s[j+1] = iks then k=j+2 else k=j)
  lemma RaEq: forall k k1 base1 base2: int.
      radix js0 k base1 /\ radix js0 k1 base2 -> k = k1 /\ base1 = base2
  lemma RaEx: exists k, base1: int. radix js0 k base1
  constant ks: int
  constant base: int
  axiom Kse: radix js0 ks base
  lemma Lba: isBase base
  lemma Lks: ks <= sL
  lemma Lks0: js0 <= ks

  predicate digitB(c: char) (base valD: int)
  axiom Adig:  forall c: char, base valD: int. isBase base ->
            ( 220<=c<220+base <->  digitB c base valD /\ c = valD + 220)
  predicate  digit(c: char) (valD: int) = digitB c base valD
  lemma LA0:   forall valD: int. not (digit zerO valD)
  lemma LAnul: digit nul 0
  lemma LAv:   forall c: char, base valD: int. digitB c base valD -> 0<=valD<base
  predicate isDigit(c: char) = exists valD: int. digit c valD
  predicate isDigit16(c: char) = exists valD: int. digitB c 16 valD

  predicate endNum(n: int) =
     n >= ks /\ not (isDigit s[n]) /\ (forall j: int. ks<=j<n -> isDigit s[j])
  lemma LnuEq: forall n1 n2: int. endNum n1 /\ endNum n2 -> n1 = n2
  lemma LnuEx: exists n: int. endNum n
  constant  n: int
  axiom Anu: endNum n
  lemma LnuLs: n <= sL
  lemma LnuKs: ks <= n

  inductive numRes int int int int =
      | Last: forall k m res0: int. k = m -> numRes k m res0 res0
      | Next: forall k m res0 res valD: int.
              k <= m /\ digit s[m] valD /\ numRes k m res0 res ->
                          numRes k (m+1) res0 (res * base + valD)
(*    | Next: forall k m res0 res valD: int.
              k < m /\ digit s[k] valD /\ numRes k m res0 res ->
                           numRes (k+1) m (res0 * base + valD) res *)
  lemma NuEq: forall k m res0 res res1: int.
                 numRes k m res0 res /\ numRes k m res0 res1 -> res1 = res
  lemma NuNext: forall res res0 k m valD: int. k < m /\ digit s[k] valD /\
                  (forall j: int. k<=j<m -> isDigit s[j]) /\
                   numRes (k+1) m (res0 * base + valD) res -> numRes k m res0 res
  predicate number(res: int) = numRes ks n 0 res
(*  lemma NuEx: exists res: int. number res *)
  constant resN: int
  axiom Ares: number resN
  predicate afterNum() = ks<n /\ (s[n]= zerO \/ s[n]=nl /\ s[n+1]= zerO)
end (*Base*)

theory Kstrtoull
  use Base
  use export int.EuclideanDivision
  use export bv.Pow2int

  constant max64: int = pow2 64 - 1        (*= ULLONG_MAX*)
  predicate pValid() = afterNum() /\ resN <= max64
  predicate pERange() = resN > max64
  lemma Lpre12: not (pValid() /\ pERange())
(*  predicate pESint() =  not (isDigit s[ks]) \/ not afterNum()*)
  predicate pESint() =  not pValid() /\ not pERange()
  lemma Lpre3: not (pValid() /\ pESint())
  predicate qValid(res: int) = isDigit s[ks] /\ res = resN
  predicate qkstrtoull( res e: int) =
     (e = 0 -> pValid() /\ qValid res) /\
     (e = 1 -> pERange()) /\
     (e = 2 -> pESint())
  predicate sS( js: int) = if s[0]=plu then js = 1 else js = 0
  goal COR1: forall res e: int. sS(js0) /\ qValid(res) /\ pValid() /\ e=0 -> qkstrtoull res e
  goal COR2: forall res e: int. sS(js0) /\ pERange() /\ e=1 -> qkstrtoull res e
  goal COR3: forall res e: int. sS(js0) /\ pESint() /\ e=2 -> qkstrtoull res e
```

```
end (*Kstrtoull*)

theory Kstrtoull_
  use Base
  use Kstrtoull
(*  predicate p_kstr(js: int) = js=js0*)
  predicate q_kstr(res c: int) =
     (c = 0 -> pValid() /\ qValid res) /\
     (c = 1 -> pERange()) /\
     (c = 2 -> pESint())
  predicate qParse(p: int)(rv: nat32)(of: bool) =
     rv = n - ks /\ of = (p>max64) /\ p = resN
  predicate p3(_res: int)(rv: nat32)(of: bool) = radix js0 ks base /\ qParse _res rv of
  goal COR4: forall res _res c: int, rv: nat32, of: bool.
     p3 _res rv of /\ of /\ c=1 -> q_kstr res c
  goal COR5: forall res _res c: int, rv: nat32, of: bool.
     p3 _res rv of /\ not of /\ rv = 0 /\ c=2 -> q_kstr res c
  predicate p4(js2 _res: int)(rv: nat32)(of: bool) =
     p3 _res rv of /\ not of /\ rv <> 0 /\ js2 = ks + rv
  predicate s3(js2 js3: int) = if s[js2] = nl then js3 = js2+1 else js3 = js2
  goal COR6: forall res _res c js2 js3: int, rv: nat32, of: bool.
     p4  js2 _res rv of /\ s3 js2 js3 /\ s[js3] <> zerO /\ c=2 -> q_kstr res c
  goal COR7: forall res _res c js2 js3: int, rv: nat32, of: bool.
     p4  js2 _res rv of /\ s3 js2 js3 /\ s[js3] = zerO /\ res = _res /\ c=0  -> q_kstr res c
end (*Kstrtoull_*)

theory Parse_integer_fixup_radix
  use Base
  use Kstrtoull
  use Kstrtoull_

  predicate s4() =
     if base0 = 0 then  ( if s[js0] = nul then
                                    ( if s[js0+1] = iks then base = 16 else base = 8 )
                              else  base = 10 )
     else base = base0
(*  goal QS1: forall js: int. p_kstr js -> s4() *)
  predicate c2() = base = 16 /\ s[js0] = nul /\ s[js0+1] = iks
  goal COR8: forall js: int.
     s4() /\ c2() /\ js= js0 + 2 -> radix js0 js base
  goal COR9: forall js: int.
     s4() /\ not c2() /\ js= js0 -> radix js0 js base
end  (*Parse_integer_fixup_radix*)

theory Parse_integer
  use Base
  use Kstrtoull
  use Kstrtoull_
  predicate pParseInt(js res rv: int)(of0: bool) =
    ks<=js<=n /\ (forall j: int. ks<=j<js -> isDigit s[j]) /\
    numRes ks js 0 res /\ rv = js-ks /\ of0 = (res>max64)
  goal QSB2: pParseInt ks 0 0 false
  predicate qParseInt(js res res9: int)(rv rv9: nat32)(of: bool) =
     numRes js n res res9  /\ rv9 = rv + n - js /\ of = (res9 > max64)
  function m(js: int): int = sL - js
  goal COR10: forall p res: int, rv: nat32, of: bool.
     qParseInt ks 0 res 0 rv of /\ p = res -> qParse p rv of
  lemma JsEqN: forall js:int. js>=ks /\ (forall j:int. ks<=j<js -> isDigit s[j]) /\
                   not (isDigit s[js])  -> js = n
  goal RP1: forall js res res9: int, rv rv9: nat32, of of0: bool.
     pParseInt js res rv of0 /\ not (isDigit s[js]) /\ res9 = res /\ rv9 = rv /\ of=of0 ->
          qParseInt js res res9 rv rv9 of
  predicate e6(res valD: int)(of0 of1: bool)  =
     if res >= pow2 60 /\ res > div (max64 - valD) base then of1 = true else of1 = of0

  lemma RnumR: forall js:int, valD:int, res:int.
     js <= n /\ (forall j:int. ks <= j < js -> isDigit s[j]) /\
     numRes ks js 0 res /\ digit s[js] valD  ->
                   numRes ks (js + 1) 0 ((res * base) + valD)
  lemma Rgr60: forall valD:int, res:int.
               0<=valD<base /\ max64 < ((res * base) + valD) -> res >= pow2 60
  predicate p6(js res valD rv: int)(of0 of1: bool) = pParseInt js res rv of0 /\
      isDigit s[js] /\ digit s[js] valD /\  e6 res valD of0 of1
  goal RB1: forall js valD res: int, rv: nat32, of0 of1: bool. p6 js res valD rv of0 of1 ->
                         pParseInt (js+1) (res*base+valD) (rv+1) of1 /\ m(js+1) < m(js)
  lemma Rlsn: forall js: int. js<=n /\ isDigit(s[js]) -> js<n
  goal RB2: forall js valD res res9: int, rv rv9: nat32, of of0 of1: bool.
     p6 js res valD rv of0 of1 /\ qParseInt (js+1) (res*base+valD) res9 (rv+1) rv9 of ->
          qParseInt js res res9 rv rv9 of
end (*Parse_integer*)

theory LemmaFunctions
  use Base
  use Kstrtoull
(*correctness formulas of lemma function for the lemma:
  lemma NuNext: forall res:int, res0:int, k:int, m:int, valD:int.
   k < m /\ digit s[k] valD /\ numRes (k + 1) m ((res0 * base) + valD) res ->
                      numRes k m res0 res
*)
  predicate pNun(k m valD res0 res: int) =
      k < m /\ digit s[k] valD /\ (forall j: int. k <=j< m -> isDigit s[j])
  predicate qNun(k m valD res0 res: int) =
            numRes (k + 1) m ((res0 * base) + valD) res -> numRes k m res0 res
  goal COR11: forall k m valD res0 res: int.
               pNun k m valD res0 res  /\ m = k+1 -> qNun k m valD res0 res
  goal QSB3:  forall k m valD res0 res res1: int.
```

```
                pNun k m valD res0 res  /\ m <> k+1 ->
                      pNun k (m-1) valD res0 res1 /\ isDigit s[m-1]
  goal COR12: forall k m valD va res0 res res1: int.
           pNun k m valD res0 res /\ m <> (k+1) /\ qNun k (m-1) valD res0 res1 /\
                  digit s[m-1] va /\ res = ((res1*base)+va) -> qNun k m valD res0 res

(* correctness formulas of lemma function for the lemma:
   lemma NuEq: forall k m res0 res res1: int.
                     numRes k m res0 res /\ numRes k m res0 res1 -> res1 = res
*)
  predicate qNuEq(k m res res0 res1: int) =
         numRes k m res0 res /\ numRes k m res0 res1 -> res1 = res
  goal COR13: forall k m res0 res res1: int. k = m -> qNuEq k m res res0 res1
  goal RB5: forall k m res0 res res1: int.
              k <> m /\ qNuEq k (m-1) res res0 res1 -> qNuEq k m res res0 res1

end (*LemmaFunctions*)
```

Приложение 3
Генерация формул корректности

Приведем набор определений, необходимых для верификации программы kstrtoull и используемых в предусловиях и постусловиях.

```
char zero = '\0', nul = '0', nl = "\n", iks = "x", plu= "+";
formula isBase(nat base) = base = 8 or base = 10 or base = 16;
type arCh = array(char, nat);
arCh s; (*первый аргумент kstrtoull *)
nat  sL; (* индекс последней литеры исходной строки s*)
axiom ALnat: sL >= 0;
axiom ALs: s[sL] = zero;
nat base0; (*второй аргумент kstrtoull *)
axiom Ab0: base0 = 0 or isBase(base0);
nat js0 = if s[0] = plu then 1 else 0;
formula radix(nat j, k, base) =
          (if base0=0 then
             (if s[j] = '0' then (if s[j+1]='x' then base = 16  else  base =8)
              else base =10 )
           else base = base0)
          &
          (if base=16 & s[j] = '0' & s[j+1='x' then k=j+2 else k=j);
nat  ks; (* индекс начальной цифры в исходной строке s*)
nat base; (*итоговое *)
axiom Kse: radix(js0, ks, base);
formula digitB(char c, nat base, valD);
axiom Adig:  forall char c, nat base, valD. isBase(base) ⇒
           ( 220<=c<220+base <->  digitB(c, base, valD) /\ c = valD + 220)
formula  digit(char c, nat valD) = digitB(c, base, valD);
formula isDigit(char c) = ∃nat valD. digit(c, valD);
formula isDigit16(char c) = ∃nat valD. digitB(c, 16, valD);
formula endNum(nat n) = n >= ks & ¬isDigit(s[n]) & ∀j=ks..n-1. isDigit(s[j]);
nat  n; (* индекс за последней цифрой в строке s*)
axiom Anu: endNum(n);
formula numRes(nat k, m, nat res0, res) =
          if (k= m) res = res0
          else ∃val. digit(s[k], val) & numRes(k+1, res0 * base + val, res);
formula number(nat res) = numRes(k, 0, res);
nat resN;
axiom Ares: number(resN);
formula afterNum() = s[n]= zero \/ s[n]=nl /\ s[n+1]= zero;
```

Формулы корректности программы **kstrtoull**

```
nat max64 = 2**64 – 1; // = ULLONG_MAX
formula pValid() = isDigit(s[ks]) & afterNum() & resN <= max64;
formula pERange() = isDigit(s[ks]) & resN > max64;
lemma Lpre12:  ¬ (pValid() & pERange())
formula pESint() = not pValid() /\ not pERange();
lemma Lpre3: not (pValid() /\ pESint());
formula qValid(nat64 res) =  isDigit(s[ks]) & res = resN

kstrtoull( : nat64 res #Valid : #ERange : #ESint)
pre Valid: pValid() pre ERange: pERange() pre ESint: pESint()
post Valid: qValid(res)
{       if (s[0] == '+') js0 = 1 else js0 =0;
        _kstrtoull( : res #Valid : #ERange : #ESint);
}
```

Построим правила корректности для гиперфункции kstrtoull.

Для гиперфункции вводим параметр-результат e – номер ветви гиперфункции со значениями 0, 1 и 2. Постусловие для гиперфункции:

```
formula qkstrtoull(nat64 res, nat e) =
        (e = 0 ⇒ pValid() & qValid(res)) &
        (e = 1 ⇒ pERange()) &
        (e = 2 ⇒ pESint())
```

Применим правило **QS**.

$$\mathbf{QS}:\ \frac{\begin{array}{l}\mathrm{P}(x) \Rightarrow \exists z.\ \mathrm{B}(x\!:\ z);\\ \forall z.\ \mathrm{C}(x,\ z\!:\ y)\ \mathbf{corr}\ [\mathrm{P}(x)\ \&\ \mathrm{B}(x\!:\ z),\ \mathrm{Q}(x,\ y)];\end{array}}{\mathrm{B}(x\!:\ z);\ \mathrm{C}(x,\ z\!:\ y)\ \mathbf{corr}\ [\mathrm{P}(x),\ \mathrm{Q}(x,y)]}$$

Конкретизация правила:

$$\mathbf{QS}:\ \frac{\begin{array}{l}\exists\ \text{js. }\mathbf{if}\ \text{(s[0] = '+') js0 = 1 }\mathbf{else}\text{ js0 =0;}\\ \text{\_kstrtoull(: res \#Valid : \#ERange : \#ESint) }\mathbf{corr}\\ \text{[S(js0), qkstrtoull(res, e)];}\end{array}}{\begin{array}{l}\mathbf{if}\text{ (s[0] = '+') js0 = 1 }\mathbf{else}\text{ js0 =0; \_kstrtoull(: res \#Valid : \#ERange : \#ESint) }\mathbf{corr}\\ \text{[}\mathbf{true}\text{, qkstrtoull(res, e)]}\end{array}}$$

```
formula S(size_t js) = (s[0] = '+' ⇒ js =1) & (¬s[0] = '+' ⇒ js =0);
```

Вторая посылка правила определяет новую цель:

```
_kstrtoull(  : res #Valid : #ERange : #ESint) corr [S(js0), qkstrtoull( res, e)]
```

Применим правило **HSB** [27, Раздел 7].

```
       B(x: y, z, e) corr* [P(x), Q(x, y, z, e)];      P1(x) ⇒ P*(x);
       C(y: u) corr [P1(x) & S(x, y) & E(x), Q1(x,v,u)];
       D(z: u) corr [P1(x) & R(x, z) & ¬E(x), Q1(x,v,u)]
HSB:  ---------------------------------------------------------------------------
       B(x: y #M1: z #M2) case M1: C(y: u) case M2: D(z: u)  corr [P1(x), Q1(x,v,u)]
```

Здесь B – гиперфункция вида

B(x: y #1: z #2) **pre** P(x) **pre** 1**:** E(x) **post** 1: S(x, y) **post** 2: R(x, z){ … };

Вызов гиперфункции преобразуется к виду:

_kstrtoull(**:** res #M1 **:** #M2 **:** #M3) **case** M1:{e=0} **case** M2: {e=1} **case** M3:{e=2}

Определим предусловие и постусловие для _kstrtoull.

```
formula p_kstr(size_t js) = true;
formula q_kstr(nat64 res, nat c) =
    (c = 0 -> pValid() /\ qValid(res)) /\
    (c = 1 -> pERange()) /\
    (c = 2 -> pESint())
```

Ниже конкретизация правила **HSB**.

```
      _kstrtoull(  : res #M1 : #M2 : #M3) corr* [p_kstr(js), q_kstr(res, c)];
      S(js0) ⇒ p_kstr(js, base);
      e=0 corr [S(js0) & qValid( res) & pValid(),  qkstrtoull( res, e)];
      e=1 corr [S(js0) & pERange(), qkstrtoull( res, e)]
      e=2 corr [S(js0) & pESint(), qkstrtoull( res, e)]
HSB: ---------------------------------------------------------------------
      _kstrtoull( : res #M1 : #M2 : #M3)
      case M1:{e=0} case M2: {e=1} case M3:{e=2} corr
      [S(js0), qkstrtoull( res, e)]
```

Вторая посылка правила определяет формулу корректности:

~~**HSB2:** S(js) ⇒ p_kstr(js);~~

Для посылок 3-5 применяется правило **COR**.

```
        ∀x,y. P(x) & H(x: y) ⇒ Q(x, y);
        ∀x. P(x) ⇒ ∃y. H(x: y)
COR:  ----------------------------------
        H(x: y) corr  [P(x), Q(x, y)]
```

Конкретизация правила для трех посылок:

```
       S(js0) & qValid( res) & pValid() & e=0 ⇒  qkstrtoull( res, e);
       S(js0) & qValid( res) & pValid() ⇒ ∃ e. e=0
COR:  ------------------------------------------------------------------
       e=0 corr [S(js0) & qValid( res) & pValid(), qkstrtoull( res, e)];
```

```
       S(js0) & pERange() & e=1 ⇒  qkstrtoull( res, e);
       S(js0) & pERange() ⇒ ∃ e. e=1
COR:  ------------------------------------------------------------------
       e=1 corr [S(js0) & pERange(), qkstrtoull( res, e)]
```

$$\textbf{COR:}\ \frac{\begin{array}{l}S(js0)\ \&\ pESint()\ \&\ e=2 \Rightarrow qkstrtoull(res, e);\\ S(js0)\ \&\ pESint() \Rightarrow \exists\ e.\ e=2\end{array}}{e=2\ \textbf{corr}\ [S(js0)\ \&\ pESint(),\ qkstrtoull(res, e)];}$$

Первые посылки в каждой их трех конкретизаций дают формулы корректности:

COR1: S(js0) & qValid(res) & pValid() & e=0 ⇒ qkstrtoull(res, e);
COR2: S(js0) & pERange() & e=1 ⇒ qkstrtoull(res, e);
COR3: S(js0) & pESint() & e=2 ⇒ qkstrtoull(res, e);

Формулы корректности программы _kstrtoull

```
_kstrtoull(  : nat64 res #Valid : #ERange : #ESint)
pre Valid: pValid() pre ERange: pERange() pre ESint: pESint()
post Valid: qValid(res)
{
        _parse_integer_fixup_radix(~~js0~~: base, ks);
        _parse_integer(~~ks, base1~~: nat64 _res, nat32 rv, bool of);
        if (of) #ERange
        else if (rv == 0) #ESint
        else { size_t js2 = ks + rv;
                if (s[js2] == '\n') js3 = js2+1 else js3 = js2;
                if (s[js3] != zero) #ESint
                else {res = _res; #Valid }
        }
}

~~predicate p_kstr(js: int) = js=js0~~
formula q_kstr(nat64 res, nat c) =
    (c = 0 -> pValid() /\ qValid(res)) /\
    (c = 1 -> pERange()) /\
    (c = 2 -> pESint())
nat sizemax;
type size_t = subtype(nat x: x<= sizemax);
```

Для тела программы _kstrtoull применяется правило **QSB**.

$$\textbf{QSB:}\ \frac{\begin{array}{l}B(x:\ z)\ \textbf{corr}^*\ [P_B(x),\ Q_B(x, z)];\ P(x) \Rightarrow P^*_B(x);\\ \forall z.\ C(x, z:\ y)\ \textbf{corr}\ [P(x)\ \&\ Q_B(x, z),\ Q(x, y)];\end{array}}{B(x:\ z);\ C(x, z:\ y)\ \textbf{corr}\ [P(x),\ Q(x,y)]}$$

Конкретизация правила **QSB**.

```
        _parse_integer_fixup_radix(: base, ks) corr*
                [true, radix(js0, ks, base)];
        p_kstr(js, base) ⇒ p_kstr(js, base);
QSB:    Z corr [radix(js0, ks, base), q_kstr(res, c)];
      ------------------------------------------------------------
        _parse_integer_fixup_radix(: base, ks); Z corr
        [true, q_kstr(  res, c)]
```

Здесь и далее Z обозначает оставшуюся часть программы. Третья посылка определяет новую цель.

```
_parse_integer( : nat64 _res, nat32 rv, bool of); Z corr
                              [radix(js0, ks, base), q_kstr(res, c)];
```

Применяется правило **QSB**.

```
           B(x: z) corr* [PB(x), QB(x, z)]; P(x) ⇒ P*B(x);
           ∀z. C(x, z: y) corr [P(x) & QB(x, z), Q(x, y)];
QSB:     ----------------------------------------------------
           B(x: z); C(x, z: y) corr [P(x), Q(x,y)]
```

formula qParse(nat64 p, nat32 rv, **bool** of) =
rv = n – ks & of = (p>max64) & (p>max64 ⇒ p = resN);
_parse_integer(~~size_t~~ ~~js, base~~ **:** nat64 p, nat32 rv, **bool** of)
~~**pre** isBase(base) & js=ks~~ **post** qParse(p, rv, of)

Конкретизация правила **QSB**.

```
        _parse_integer( : _res, rv, of) corr*   [true, qParse( _res, rv, of)];
        radix(js0, ks, base) ⇒ true;
QSB:    Z corr [radix(js0, ks, base) & qParse( _res, rv, of), q_kstr(res, c)];
      ------------------------------------------------------------------------
        _parse_integer( : _res, rv, of); Z corr  [radix(js0, ks, base), q_kstr(res, c)]
```

Вторая посылка определяет формулу корректности:

~~**QSB1**: p_kstr(js) & radix(js, js1, base1) ⇒ pParse(js1, base1)~~

Третья посылка определяет новую цель.

formula p3(nat64 _res, nat32 rv, **bool** of) = radix(js0, ks, base) & qParse(_res, rv, of);
if (of) #ERange **else** Z **corr** [p3(_res, rv, of), q_kstr(res, c)];

Применяется правило **QC**.

```
          B(x: y) corr [P(x) & E(x), Q(x, y)];
          C(x: z) corr [P(x) & ¬E(x), Q(x, y)]
QC:     ----------------------------------------------
          {if (E(x)) B(x: y) else C(x: y)} corr [P(x), Q(x, y)]
```

Конкретизация правила **QC**:

```
      c=1 corr [p3( _res, rv, of) & of, q_kstr(res, c)];
      Z  corr [p3( _res, rv, of) & ¬ of, q_kstr(res, c)]
QC:  ---------------------------------------------------------------
      if (of) c=1 else Z corr [p3( _res, rv, of), q_kstr(res, c)]
```

К первой посылке применим правило **COR**.

$$\textbf{COR:}\ \frac{\begin{array}{l}\forall x,y.\ P(x)\ \&\ H(x\!:\ y) \Rightarrow Q(x, y);\\ \forall x.\ P(x) \Rightarrow \exists y.\ H(x\!:\ y)\end{array}}{H(x\!:\ y)\ \textbf{corr}\ \ [P(x), Q(x, y)]}$$

Конкретизация правила:

$$\textbf{COR:}\ \frac{\begin{array}{l}\mathrm{p3(\ \_res, rv, of)}\ \&\ \mathrm{of}\ \&\ \mathrm{c=1} \Rightarrow \mathrm{q\_kstr(res, c)};\\ \mathrm{p3(\ \_res, rv, of)}\ \&\ \mathrm{of} \Rightarrow \exists\ \mathrm{c.\ c=1}\end{array}}{\mathrm{c=1}\ \textbf{corr}\ [\mathrm{p3(\ \_res, rv, of)}\ \&\ \mathrm{of}, \mathrm{q\_kstr(res, c)}]}$$

Первая посылка определяет формулу корректности.

COR4: p3(_res, rv, of) & of & c=1 ⇒ q_kstr(res, c);

Вторая посылка правила **QC** определяет новую цель.

if (rv = 0) #ESint **else** Z **corr** [p3(_res, rv, of) & ¬ of, q_kstr(res, c)]

Применяется правило **QC**.

$$\textbf{QC:}\ \frac{\begin{array}{l}B(x\!:\ y)\ \textbf{corr}\ [P(x)\ \&\ E(x), Q(x, y)];\\ C(x\!:\ z)\ \textbf{corr}\ [P(x)\ \&\ \neg E(x), Q(x, y)]\end{array}}{\{\textbf{if}\ (E(x))\ B(x\!:\ y)\ \textbf{else}\ C(x\!:\ y)\}\ \textbf{corr}\ [P(x), Q(x, y)]}$$

Конкретизация правила **QC**:

$$\textbf{QC:}\ \frac{\begin{array}{l}\mathrm{c=2}\ \textbf{corr}\ [\mathrm{p3(\ \_res, rv, of)}\ \&\ \neg\mathrm{of}\ \&\ \mathrm{rv = 0}, \mathrm{q\_kstr(res, c)}];\\ \mathrm{Z}\ \ \textbf{corr}\ [\mathrm{p3(\ \_res, rv, of)}\ \&\ \neg\mathrm{of}\ \&\ \neg\mathrm{rv = 0}, \mathrm{q\_kstr(res, c)}]\end{array}}{\textbf{if}\ \mathrm{(rv = 0)\ c=2}\ \textbf{else}\ \mathrm{Z}\ \textbf{corr}\ [\mathrm{p3(\ \_res, rv, of)}\ \&\ \neg\ \mathrm{of}, \mathrm{q\_kstr(res, c)}]}$$

К первой посылке применим правило **COR**.

$$\textbf{COR:}\ \frac{\begin{array}{l}\forall x,y.\ P(x)\ \&\ H(x\!:\ y) \Rightarrow Q(x, y);\\ \forall x.\ P(x) \Rightarrow \exists y.\ H(x\!:\ y)\end{array}}{H(x\!:\ y)\ \textbf{corr}\ \ [P(x), Q(x, y)]}$$

Конкретизация правила:

$$\textbf{COR:}\ \frac{\begin{array}{l}\mathrm{p3(\ \_res, rv, of)}\ \&\ \neg\mathrm{of}\ \&\ \mathrm{rv = 0}\ \&\ \mathrm{c=2} \Rightarrow \mathrm{q\_kstr(res, c)};\\ \mathrm{p3(\ \_res, rv, of)}\ \&\ \neg\mathrm{of}\ \&\ \mathrm{rv = 0} \Rightarrow \exists\ \mathrm{c.\ c=2}\end{array}}{\mathrm{c=2}\ \textbf{corr}\ [\mathrm{p3(\ \_res, rv, of)}\ \&\ \neg\mathrm{of}\ \&\ \mathrm{rv = 0}, \mathrm{q\_kstr(res, c)}];}$$

Первая посылка определяет формулу корректности.

COR5: p3(_res, rv, of) & ¬of & rv = 0 & c=2 ⇒ q_kstr(res, c);

Вторая посылка правила **QC** определяет новую цель.

size_t js2 = ks + rv; Z **corr** [p3(_res, rv, of) & ¬of & ¬rv = 0, q_kstr(res, c)]

Применяется правило **QS**.

$$\textbf{QS:}\ \frac{\begin{array}{l}P(x) \Rightarrow \exists z.\ B(x\!:\ z);\\ \forall z.\ C(x, z\!:\ y)\ \textbf{corr}\ [P(x)\ \&\ B(x\!:\ z), Q(x, y)];\end{array}}{B(x\!:\ z);\ C(x, z\!:\ y)\ \textbf{corr}\ [P(x), Q(x,y)]}$$

Конкретизация правила:

```
      p3( _res, rv, of) & ¬of & ¬rv = 0 ⇒ ∃js2. js2 = ks + rv;
      Z  corr [p3( _res, rv, of) & ¬of & ¬rv = 0 & js2 = ks + rv, q_kstr(res, c)];
QS:  ───────────────────────────────────────────────────────────────────────────
      js2 = ks + rv; Z  corr [p3( _res, rv, of) & ¬of & ¬rv = 0, q_kstr(res, c)]
```

Вторая посылка определяет новую цель.

formula p4(size_t js2, nat64 _res, nat32 rv, **bool** of) =
    p3(_res, rv, of) & ¬of & ¬rv = 0 & js2 = ks + rv;
if (s[js2] == '\n') js3 = js2+1 **else** js3 = js2; Z **corr** [p4(js2, _res, rv, of), q_kstr(res, c)];
formula S3(size_t js2, js3) = **if** (s[js2] = '\n') js3 = js2+1 **else** js3 = js2;

Применяется правило **QS**.

```
      P(x) ⇒ ∃z. B(x: z);
      ∀z. C(x, z: y) corr [P(x) & B(x: z), Q(x, y)];
QS:  ──────────────────────────────────────────────
      B(x: z); C(x, z: y) corr [P(x), Q(x,y)]
```

Конкретизация правила:

```
      p4(js2,  _res, rv, of) ⇒ ∃js3. if (s[js2] = '\n') js3 = js2+1 else js3 = js2;
      Z corr [p4(js2, _res, rv, of) & S3(js2, js3), q_kstr(res, c)];
QS:  ─────────────────────────────────────────────────────────────────────────────────
      if (s[js2] == '\n') js3 = js2+1 else js3 = js2; Z corr [p4(js2, _res, rv, of), q_kstr(res, c)]
```

Вторая посылка определяет новую цель.

if (s[js3] != zero) #ESint **else** {res = _res; #Valid } **corr**
    [p4(js2, _res, rv, of) & S3(js2, js3), q_kstr(res, c)];

Применяется правило **QC**.

```
      B(x: y) corr [P(x) & E(x), Q(x, y)];
      C(x: z) corr [P(x) & ¬E(x), Q(x, y)]
QC:  ──────────────────────────────────────────────
      {if (E(x)) B(x: y) else C(x: y)} corr [P(x), Q(x, y)]
```

Конкретизация правила **QC**:

```
      c=2 corr [p4(js2, _res, rv, of) & S3(js2, js3) & s[js3] != zero, q_kstr(res, c)];
      res = _res; #Valid  corr [p4(js2, _res, rv, of) & S3(js2, js3) & s[js3] = zero, q_kstr(res, c)]
QC:  ─────────────────────────────────────────────────────────────────────────────────────────────
      if (s[js3] != zero) #ESint else {res = _res; #Valid } corr
      [p4(js2, _res, rv, of) & S3(js2, js3), q_kstr(res, c)]
```

К первой посылке применим правило **COR**.

```
       ∀x,y. P(x) & H(x: y) ⇒ Q(x, y);
       ∀x. P(x) ⇒ ∃y. H(x: y)
COR:  ─────────────────────────────────
       H(x: y) corr  [P(x), Q(x, y)]
```

Конкретизация правила:

```
       p4(js2, _res, rv, of) & S3(js2, js3) & s[js3] != zero & c=2 ⇒ q_kstr(res, c);
       p4(js2, _res, rv, of) & S3(js2, js3) & s[js3] != zero ⇒ ∃ c. c=2
COR:  ─────────────────────────────────────────────────────────────────────────────────
       c=2 corr [p4(js2, _res, rv, of) & S3(js2, js3) & s[js3] != zero, q_kstr(res, c)];
```

Первая посылка определяет формулу корректности.

COR6: p4(js2, _res, rv, of) & S3(js2, js3) & s[js3] != zero & c=2 ⇒ q_kstr(res, c);

Вторая посылка правила **QC** определяет новую цель.

res = _res; #Valid **corr** [p4(js2, _res, rv, of) & S3(js2, js3) & s[js3] = zero, q_kstr(res, c)];

Применяется правило **QS**.

$$\textbf{QS:}\ \frac{\begin{array}{l}P(x) \Rightarrow \exists z.\ B(x\textbf{:}\ z);\\ \forall z.\ C(x, z\text{: } y)\ \textbf{corr}\ [P(x)\ \&\ B(x\textbf{:}\ z), Q(x, y)];\end{array}}{B(x\text{: } z);\ C(x, z\text{: } y)\ \textbf{corr}\ [P(x), Q(x,y)]}$$

Конкретизация правила:

$$\textbf{QS:}\ \frac{\begin{array}{l}p4(js2, \_res, rv, of)\ \&\ S3(js2, js3)\ \&\ s[js3] = zero \Rightarrow \exists res.\ res = \_res;\\ c{=}0\ \textbf{corr}\ [p4(js2, \_res, rv, of)\ \&\ S3(js2, js3)\ \&\ s[js3] = zero\ \&\ res = \_res, q\_kstr(res, c)];\end{array}}{res = \_res;\ \#Valid\ \ \textbf{corr}\ [p4(js2, \_res, rv, of)\ \&\ S3(js2, js3)\ \&\ s[js3] = zero, q\_kstr(res, c)]}$$

Для второй посылки применяется **COR**.

$$\textbf{COR:}\ \frac{\begin{array}{l}\forall x,y.\ P(x)\ \&\ H(x\textbf{:}\ y) \Rightarrow Q(x, y);\\ \forall x.\ P(x) \Rightarrow \exists y.\ H(x\textbf{:}\ y)\end{array}}{H(x\textbf{:}\ y)\ \textbf{corr}\ \ [P(x), Q(x, y)]}$$

Конкретизация правила:

$$\textbf{COR:}\ \frac{\begin{array}{l}p4(js2, \_res, rv, of)\ \&\ S3(js2, js3)\ \&\ s[js3] = zero\ \&\ res = \_res\ \&\ c{=}0\ \ \Rightarrow q\_kstr(res, c);\\ p4(js2, \_res, rv, of)\ \&\ S3(js2, js3)\ \&\ s[js3] = zero\ \&\ res = \_res \Rightarrow \exists\ c.\ c{=}0\end{array}}{c{=}0\ \textbf{corr}\ [p4(js2, \_res, rv, of)\ \&\ S3(js2, js3)\ \&\ s[js3] = zero\ \&\ res = \_res, q\_kstr(res, c)]}$$

Первая посылка определяет формулу корректности.

COR7: p4(js2, _res, rv, of) & S3(js2, js3) & s[js3] = zero & res = _res & c=0 ⇒ q_kstr(res, c);

Формулы корректности программы _parse_integer_fixup_radix

```
_parse_integer_fixup_radix( : nat base, size_t js)
~~pre js=js0~~ post radix(js0, js, base)
{
        if (base0 == 0) {
                if (s[js0] == '0') {
                        if (s[js0+1]) == 'x' (*& isxdigit(s[js+2])*)) base = 16
                        else base = 8;
                } else  base = 10;
        };
        if (base == 16 & s[js0] == '0' & s[js0+1] == 'x') js= js0 + 2 else js = js0;
}
```

```
formula S4( nat base) =
     if base0 = 0 then  ( if s[js0] = '0' then
                                    ( if s[js0+1]) = 'x' then base = 16 else base = 8)
                              else  base = 10 )
     else base = base0;
```

Для тела программы _parse_integer_fixup_radix применяется правило **QS**.

$$\textbf{QS:}\ \frac{\begin{array}{l}\text{P(x)} \Rightarrow \exists\text{z. B(x: z);}\\ \forall\text{z. C(x, z: y) }\textbf{corr}\text{ [P(x) \& B(x: z), Q(x, y)];}\end{array}}{\text{B(x: z); C(x, z: y) }\textbf{corr}\text{ [P(x), Q(x,y)]}}$$

Конкретизация правила:

$$\textbf{QS:}\ \frac{\begin{array}{l}\exists\text{ base. S4(base);}\\ \text{Z }\textbf{corr}\text{ [S4(base), radix(js0, js, base)];}\end{array}}{\textbf{if}\text{ (base0 == 0)…; Z }\textbf{corr}\text{ [}\textbf{true}\text{, radix(js0, js, base)]}}$$

Первая посылка определяет формулу корректности:

QS1: ∃ base. S4(base);

Для второй посылки применяется **QC**.

$$\textbf{QC:}\ \frac{\begin{array}{l}\text{B(x: y) }\textbf{corr}\text{ [P(x) \& E(x), Q(x, y)];}\\ \text{C(x: z) }\textbf{corr}\text{ [P(x) \& }\neg\text{E(x), Q(x, y)]}\end{array}}{\text{\{}\textbf{if (}\text{E(x)) B(x: y) }\textbf{else}\text{ C(x: y)\} }\textbf{corr}\text{ [P(x), Q(x, y)]}}$$

formula C2(**nat** base) = base = 16 & s[js0] = '0' & s[js0+1] = 'x';

Конкретизация правила **QC**:

$$\textbf{QC:}\ \frac{\begin{array}{l}\text{js= js0 + 2 }\textbf{corr}\text{ [S4(base) \& C2(base), radix(js0, js, base)];}\\ \text{js = js0 }\textbf{corr}\text{ [S4(base) \& }\neg\text{C2(base), radix(js0, js, base)];}\end{array}}{\begin{array}{l}\textbf{if}\text{ (base == 16 \& s[js0] == '0' \& s[js0+1] == 'x') js= js0 + 2 }\textbf{else}\text{ js = js0 }\textbf{corr}\\ \text{[S4(base), radix(js0, js, base)]}\end{array}}$$

К первой и второй посылке применим правило **COR**.

$$\textbf{COR:}\ \frac{\begin{array}{l}\forall\text{x,y. P(x) \& H(x: y)} \Rightarrow \text{Q(x, y);}\\ \forall\text{x. P(x)} \Rightarrow \exists\text{y. H(x: y)}\end{array}}{\text{H(x: y) }\textbf{corr}\text{ [P(x), Q(x, y)]}}$$

Конкретизации правила **COR**:

$$\textbf{COR:}\ \frac{\begin{array}{l}\text{S4(base) \& C2(base) \& js= js0 + 2} \Rightarrow \text{radix(js0, js, base);}\\ \text{S4(base) \& C2(base)} \Rightarrow \exists\text{ js'. js= js0 + 2}\end{array}}{\text{js= js0 + 2 }\textbf{corr}\text{ [S4(base) \& C2(base), radix(js0, js, base)];}}$$

$$\textbf{COR:}\ \frac{\begin{array}{l}\text{S4(base) \& }\neg\text{C2(base) \& js = js0} \Rightarrow \text{radix(js0, js, base);}\\ \text{S4(base) \& }\neg\text{C2(base)} \Rightarrow \exists\text{ js. js = js0}\end{array}}{\text{js = js0 }\textbf{corr}\text{ [S4(base) \& }\neg\text{C2(base), radix(js0, js, base)];}}$$

Первые посылки в каждой из конкретизаций дают формулы корректности:

COR8: S4(base) & C2(base) & js= js0 + 2 ⇒ radix(js0, js, base);
COR9: S4(base) & ¬C2(base) & js = js0 ⇒ radix(js0, js, base);

Формулы корректности программы _parse_integer

formula qParse(nat64 p, nat32 rv, **bool** of) =
rv = n – ks & of = (p>max64) & p = resN;
_parse_integer(**:** nat64 p, nat32 rv, **bool** of)
~~**pre** isBase(base) & js=ks~~ **post** qParse(p, rv, of)
{ parseInt(ks, 0, 0, **false: ** nat64 res, rv, of);
p = res;
}

Для тела программы _parse_integer применяется правило **QSB**.

$$\mathbf{QSB}:\ \frac{\begin{array}{l}B(x\!: z)\ \mathbf{corr}^*\ [P_B(x), Q_B(x, z)];\ P(x) \Rightarrow P^*_B(x);\\ \forall z.\ C(x, z\!: y)\ \mathbf{corr}\ [P(x)\ \&\ Q_B(x, z), Q(x, y)];\end{array}}{B(x\!: z);\ C(x, z\!: y)\ \mathbf{corr}\ [P(x), Q(x,y)]}$$

Конкретизация правила **QSB**.

$$\mathbf{QSB}:\ \frac{\begin{array}{l}\text{parseInt(ks, 0, 0, } \mathbf{false}\mathbf{:}\ \text{nat64 res, rv, of) } \mathbf{corr}^*\\ \text{[pParseInt(ks, 0, 0, } \mathbf{false}\text{), qParseInt(ks, 0, res, 0, rv, of)];}\\ \text{pParseInt(ks, 0, 0, } \mathbf{false}\text{);}\\ \text{p = res } \mathbf{corr}\text{ [qParseInt(ks, 0, res, 0, rv, of), qParse(p, rv, of)];}\end{array}}{\text{parseInt(ks, 0, 0, } \mathbf{false}\mathbf{:}\ \text{nat64 res, rv, of); p = res } \mathbf{corr}\ [\mathbf{true}\text{, qParse(p, rv, of)]}}$$

Вторая посылка определяет формулу корректности:

QSB2: pParseInt(ks, 0, 0, **false**);

Для третьей посылки применяется правило **COR**.

$$\mathbf{COR}:\ \frac{\begin{array}{l}\forall x,y.\ P(x)\ \&\ H(x\mathbf{:}\ y) \Rightarrow Q(x, y);\\ \forall x.\ P(x) \Rightarrow \exists y.\ H(x\mathbf{:}\ y)\end{array}}{H(x\mathbf{:}\ y)\ \mathbf{corr}\ \ [P(x), Q(x, y)]}$$

Конкретизация правила **COR**:

$$\mathbf{COR}:\ \frac{\begin{array}{l}\text{qParseInt(ks, 0, res, 0, rv, of) \& p = res} \Rightarrow \text{qParse(p, rv, of)}\\ \text{qParseInt(ks, 0, res, 0, rv, of)} \Rightarrow \exists\ \text{p. p = res}\end{array}}{\text{p = res } \mathbf{corr}\text{ [qParseInt(ks, 0, res, 0, rv, of), qParse(p, rv, of)]}}$$

Первая посылка дает формулу корректности:

COR10: qParseInt(ks, 0, res, 0, rv, of) & p = res ⇒ qParse(p, rv, of);

Построим формулы корректности для программы parseInt.

formula m(size_t js) = sL – js;
formula pParseInt(size_t js, nat64 res, nat32 rv, **bool** of0) =
ks<=js<=n & (∀**nat** j=ks..js-1. isDigit(s[j])) &
numRes(ks, js, 0, res) & rv = js-ks & of0 =(res>max64);
formula qParseInt(size_t js, nat64 res, res’, nat32 rv, rv’, of) =
numRes(js, n, res, res’) & rv’ = rv + n – js & of = res’>max64;

parseInt (size_t js, nat64 res, nat32 rv, **bool** of0**: nat** res’, nat32 rv’, **bool** of)

```
pre pParseInt(js, res, rv, of0) post qParseInt(js, res, res', rv, rv', of) measure sL – js;
{       if (isDigit(s[js])) {
                digit(s[js], val);
                ~~nat res1 = res*base + val;~~
                if (res >= 2**60 & res > (max64 – val) / base) of1= true else of1 = of0;
                parseInt (js+1, res*base + val, rv+1, of1 :  res', rv', of);
        } else { res' = res || rv' = rv || of = of0 }
}
```

Для тела программы parseInt применяется правило **QC**.

$$\mathbf{QC}: \frac{\begin{array}{l} B(x\!: y)\ \mathbf{corr}\ [P(x)\ \&\ E(x),\ Q(x, y)]; \\ C(x\!: z)\ \mathbf{corr}\ [P(x)\ \&\ \neg E(x),\ Q(x, y)] \end{array}}{\{\mathbf{if}\ (E(x))\ B(x\!: y)\ \mathbf{else}\ C(x\!: y)\}\ \mathbf{corr}\ [P(x),\ Q(x, y)]}$$

Конкретизация правила **QC**:

$$\mathbf{QC}: \frac{\begin{array}{l} Z1\ \mathbf{corr}\ \ [pParseInt(js, res, rv, of0)\ \&\ isDigit(s[js]),\ \ qParseInt(js, res, res', rv, rv', of)]; \\ Z2\ \ \mathbf{corr}\ \ [pParseInt(js, res, rv, of0)\ \&\ \neg isDigit(s[js]),\ \ qParseInt(js, res, res', rv, rv', of)]; \end{array}}{\mathbf{if}\ (isDigit(s[js]))\ \ Z1\ \mathbf{else}\ \ Z2\ \mathbf{corr}\ \ [pParseInt(js, res, rv, of0),\ qParseInt(js, res, res', rv, rv', of)]}$$

Посылки правила **QC** определяют две новые цели.

digit(s[js], val);Z1 **corr**
[pParseInt(js, res, rv, of0) & isDigit(s[js]), qParseInt(js, res, res', rv, rv', of)]
res' = res || rv' = rv || of = of0 **corr**
        [pParseInt(js, res, rv, of0) & ¬isDigit(s[js]), qParseInt(js, res, res', rv, rv', of)]

Для второй цели применим правило **RP**.

$$\mathbf{RP}: \frac{\begin{array}{l} B(x\!: y)\ \mathbf{corr^*}\ [P_B(x),\ Q_B(x, y)]; \\ C(x\!: z)\ \mathbf{corr^*}\ [P_C(x),\ Q_C(x, z)]; \\ \forall y, z\ (P(x)\ \&\ Q_B(x, y)\ \&\ Q_C(x, z) \Rightarrow Q(x, y, z)); \\ P(x) \Rightarrow P^*_B(x)\ \&\ P^*_C(x); \end{array}}{\{B(x\!: y)\ ||\ C(x\!: z)\}\ \mathbf{corr}\ [P(x),\ Q(x, y, z)]}$$

Конкретизация правила **RP**:

$$\mathbf{RP}: \frac{\begin{array}{l} res' = res\ \mathbf{corr^*}\ [\mathbf{true},\ res' = res]; \\ rv' = rv\ \mathbf{corr^*}\ [\mathbf{true},\ rv' = rv]; \\ of = \mathbf{false}\ \mathbf{corr^*}\ [\mathbf{true},\ of = of0]; \\ pParseInt(js, res, rv, of0)\ \&\ \neg isDigit(s[js])\ \&\ res' = res\ \&\ rv' = rv\ \&\ of = of0 \Rightarrow \\ \qquad qParseInt(js, res, res', rv, rv', of); \\ pParseInt(js, res, rv, of0)\ \&\ \neg isDigit(s[js]) \Rightarrow true\ \&\ true\ \&\ true; \end{array}}{\begin{array}{l} \{\ res' = res\ ||\ rv' = rv\ ||\ of = of0\ \}\ \mathbf{corr} \\ pParseInt(js, res, rv, of0)\ \&\ \neg isDigit(s[js]),\ \ qParseInt(js, res, res', rv, rv', of)] \end{array}}$$

Четвертая посылка определяет формулу корректности:

RP1: pParseInt(js, res, rv, of0) & ¬isDigit(s[js]) & res' = res & rv' = rv & of = of0 ⇒
                qParseInt(js, res, res', rv, rv', of);

Для первой цели правила **QC**: применим правило **QS**.

$$\mathbf{QS}:\ \frac{\begin{array}{l} P(x) \Rightarrow \exists z.\ B(x\!: z); \\ \forall z.\ C(x, z\!: y)\ \mathbf{corr}\ [P(x)\ \&\ B(x\!: z), Q(x, y)]; \end{array}}{B(x\!: z);\ C(x, z\!: y)\ \mathbf{corr}\ [P(x), Q(x,y)]}$$

Конкретизация правила:

$$\mathbf{QS}:\ \frac{\begin{array}{l} \text{pParseInt(js, res, rv, of0) \& isDigit(s[js])} \Rightarrow \exists\ \text{val. digit(s[js], val);} \\ \text{Z1 } \mathbf{corr}\ \text{[pParseInt(js, res, rv, of0) \& isDigit(s[js]) \& digit(s[js], val),} \\ \qquad\qquad \text{qParseInt(js, res, res′, rv, rv′, of)];} \end{array}}{\begin{array}{l} \text{digit(s[js], val);Z1 } \mathbf{corr} \\ \text{[pParseInt(js, res, rv, of0) \& isDigit(s[js]), qParseInt(js, res, res′, rv, rv′, of)]} \end{array}}$$

Вторая посылка определяет новую цель:

E6(res, val, of0, of1); parseInt (js+1, res*base + val, rv+1, of1 **:** res′, rv′, of) **corr**
[pParseInt(js, res, rv, of0) & isDigit(s[js], 16) & digit(s[js], 16, val),
qParseInt(js, res, res′, rv, rv′, of)];

formula E6(**nat** res, val, **bool** of0, of1) =
if (res >= 2**60 & res > (max64 – val) / base) **then** of1= **true** **else** of1 = of0

Снова применяется правило **QS**.

$$\mathbf{QS}:\ \frac{\begin{array}{l} P(x) \Rightarrow \exists z.\ B(x\!: z); \\ \forall z.\ C(x, z\!: y)\ \mathbf{corr}\ [P(x)\ \&\ B(x\!: z), Q(x, y)]; \end{array}}{B(x\!: z);\ C(x, z\!: y)\ \mathbf{corr}\ [P(x), Q(x,y)]}$$

Конкретизация правила:

$$\mathbf{QS}:\ \frac{\begin{array}{l} \text{pParseInt(js, res, rv, of0) \& isDigit(s[js], 16) \& digit(s[js], 16, val)} \Rightarrow \exists\ \text{of1. E6(res, val, of0, of1)} \\ \text{parseInt (js+1, res*base + val, rv+1, of1 : res′, rv′, of) } \mathbf{corr} \\ \text{[pParseInt(js, res, rv, of0) \& isDigit(s[js], 16) \& digit(s[js], 16, val) \& E6(res, val, of0, of1),} \\ \qquad\qquad \text{qParseInt(js, res, res′, rv, rv′, of)];} \end{array}}{\begin{array}{l} \text{E6(res, val, of0, of1); parseInt (js+1, res*base + val, rv+1, of1 : res′, rv′, of) } \mathbf{corr} \\ \text{[pParseInt(js, res, rv, of0) \& isDigit(s[js], 16) \& digit(s[js], 16, val),} \\ \qquad \text{qParseInt(js, res, res′, rv, rv′, of)];} \end{array}}$$

Вторая посылка определяет новую цель:

parseInt (js+1, res*base + val, rv+1, of1 **:** res′, rv′, of) **corr**
[pParseInt(js, res, rv, of0) & isDigit(s[js], 16) & digit(s[js], 16, val) & E6(res, val, of0, of1),
qParseInt(js, res, res′, rv, rv′, of)];

применим правило **RB**:

$$\mathbf{RB}:\ \frac{\begin{array}{l} \forall z\ C(x, z\!: y)\ \mathbf{corr^*}\ [P_C(x, z), Q_C(x, y)]; \\ SV(P_B, B)(x); \\ P(x) \Rightarrow P_B(x)\ \&\ P^*_C(x, B(x)); \\ \forall y\ (\ P(x)\ \&\ Q_C(B(x), y) \Rightarrow Q(x, y)\ ); \end{array}}{C(x, B(x)\!: y)\ \mathbf{corr}\ [P(x), Q(x, y)]}$$

formula p6(size_t js, **nat** res, val, nat32 rv, **bool** of0, of1) =
pParseInt(js, res, rv, of0) & isDigit(s[js]) & digit(s[js], val) & E6(res, val, of0, of1);

```
formula pB(size_t js, nat rv) = rv+1 < 2**32;
```

Конкретизация правила:

```
    parseInt(js, res, rv, of0: res′, rv′, of) corr*
    [pParseInt(js, res, rv, of0), qParseInt(js , res, res′, rv, rv′, of)];
    p6(js, res, val, rv, of0, of1) ⇒ pB(rv) & pParseInt(js+1, res*base + val, rv+1, of1) & m(js+1)<m(js);
    p6(js, res, val, rv, of0, of1) & qParseInt(js+1, res*base + val, res′, rv1+1, rv′, of) ⇒
       qParseInt(js , res, res′, rv, rv′, of);
RB: ------------------------------------------------------------------------------------------------------
    parseInt (js+1, res*base + val, rv+1, of1 :  res′, rv′, of) corr
     [p6(js, res, val, rv, of0, of1), qParseInt(js , res, res′, rv, rv′, of)]
```

Здесь B(js, res, rv) = (js+1, res * base + val, rv1+1). Причем P_B(x) = ~~js+1 < 2**32 &~~ ~~=~~ rv+1 < 2**32. Вторая и третья посылки определяют следующие формулы корректности:

```
RB1: p6(js, res, val, rv, of0, of1) ⇒ pParseInt(js+1, res*base + val, rv+1, of1) & m(js+1)<m(js);
RB2: p6(js, res, val, rv, of0, of1) & qParseInt(js+1, res*base + val, res′, rv1+1, rv′, of) ⇒
             qParseInt(js, res, res′, rv, rv′, of)
```

Генерация формул корректности для лемм-предикатов

B.1. Лемма **NuNext.**

При доказательстве возникла следующая лемма:

```
lemma NuNext: forall res:int, res0:int, k:int, m:int, valD:int.
  k < m /\ digit s[k] valD /\ numRes (k + 1) m ((res0 * base) + valD) res ->
                  numRes k m res0 res
```

Построим следующую соответствующую программу в качестве леммы-предиката:

```
NuNext(int k, m, valD, res0, res)
      pre k < m /\ digit s[k] valD /\ numRes (k + 1) m ((res0 * base) + valD) res
      post numRes k m res0 res
      measure m
{   if (m = k+1) true
    else  {{ NuNext(k, m-1, valD, res0, res1) || digit(s[m-1], va)}; res= res1*base+va}
}

formula pNun(k, m, valD, res0, res) =
          k < m /\ digit s[k] valD /\ numRes (k + 1) m ((res0 * base) + valD) res
formula qNun(k, m, valD, res0, res) = numRes k m res0 res

   formula hm(nat m) = m;
```

Применим правило **QC**.

```
          B(x: y) corr [P(x) & E(x), Q(x, y)];
          C(x: z) corr [P(x) & ¬E(x), Q(x, y)]
QC:   ------------------------------------------------
          {if (E(x)) B(x: y) else C(x: y)} corr [P(x), Q(x, y)]
```

Конкретизация правила **QC**:

$$\textbf{QC:}\ \dfrac{\begin{array}{l}\textbf{true corr}\ \text{[pNun(k, m, valD, res0, res) \& m = k+1, qNun(k, m, valD, res0, res)];}\\ \text{Z; res= res1*base+va}\ \textbf{corr}\\ \text{[pNun(k, m, valD, res0, res) \& m <> k+1, qNun(k, m, valD, res0, res)]}\end{array}}{\textbf{if}\ \text{(m = k+1)}\ \textbf{true}\ \ \textbf{else}\ \text{Z}\ \textbf{corr}\ \text{[pNun(k, m, valD, res0, res), qNun(k, m, valD, res0, res)]}}$$

Для первой посылки применим правило **COR**.

$$\textbf{COR:}\ \dfrac{\begin{array}{l}\forall x,y.\ P(x)\ \&\ H(x\mathbf{:}\ y) \Rightarrow Q(x, y);\\ \forall x.\ P(x) \Rightarrow \exists y.\ H(x\mathbf{:}\ y)\end{array}}{H(x\mathbf{:}\ y)\ \textbf{corr}\ \ [P(x), Q(x, y)]}$$

Конкретизация правила:

$$\textbf{COR:}\ \dfrac{\begin{array}{l}\text{pNun(k, m, valD, res0, res) \& m = k+1 \& }\textbf{true} \Rightarrow \text{qNun(k, m, valD, res0, res);}\\ \text{pNun(k, m, valD, res0, res) \& m = k+1} \Rightarrow \exists\text{b. }\textbf{true}\end{array}}{\textbf{true corr}\ \text{[pNun(k, m, valD, res0, res) \& m = k+1, qNun(k, m, valD, res0, res)]}}$$

Первая посылка определяет формулу корректности:

COR11: pNun(k, m, valD, res0, res) & m = k+1 $\Rightarrow$ qNun(k, m, valD, res0, res);

Вторая посылка правила **QC** определяет цель:

Z; res= res1*base+va **corr**
[pNun(k, m, valD, res0, res) & m <> k+1, qNun(k, m, valD, res0, res)]

Применяется правило **QSB**.

$$\textbf{QSB:}\ \dfrac{\begin{array}{l}B(x\text{: }z)\ \textbf{corr}^*\ [P_B(x), Q_B(x, z)];\ P(x) \Rightarrow P^*{}_B(x);\\ \forall z.\ C(x, z\text{: }y)\ \textbf{corr}\ [P(x)\ \&\ Q_B(x, z), Q(x, y)];\end{array}}{B(x\text{: }z);\ C(x, z\text{: }y)\ \textbf{corr}\ [P(x), Q(x,y)]}$$

Конкретизация правила:

$$\textbf{QSB:}\ \dfrac{\begin{array}{l}\text{Z}\ \textbf{corr}\ \text{[pNun(k, m-1, valD, res0, res1) \& isDigit(s[m-1]),}\\ \qquad\text{qNun(k, m-1, valD, res0, res1) \& digit(s[m-1], va)]}\\ \text{pNun(k, m, valD, res0, res) \& m <> k+1 -> pNun(k, m-1, valD, res0, res1) \& isDigit(s[m-1])}\\ \text{res= res1*base+va}\ \textbf{corr}\\ \text{[pNun(k, m, valD, res0, res) \& m <> k+1 \& qNun(k, m-1, valD, res0, res1) \& digit(s[m-1], va),}\\ \text{qNun(k, m, valD, res0, res)];}\end{array}}{\begin{array}{l}\text{Z; res= res1*base+va}\ \textbf{corr}\\ \text{[pNun(k, m, valD, res0, res) \& m <> k+1, qNun(k, m, valD, res0, res)]}\end{array}}$$

Z **corr** [pNun(k, m-1, valD, res0, res1) & isDigit(s[m-1]),
qNun(k, m-1, valD, res0, res1) & digit(s[m-1], va)]

Вторая посылка определяет формулу корректности:

QSB3: pNun(k, m, valD, res0, res) & m <> k+1 ->
pNun(k, m-1, valD, res0, res1) & isDigit(s[m-1]);

Третья посылка определяет новую цель. Применяется правило **COR**.

$$\textbf{COR:}\ \frac{\forall x,y.\ P(x)\ \&\ H(x\!:\ y) \Rightarrow Q(x, y);\quad \forall x.\ P(x) \Rightarrow \exists y.\ H(x\!:\ y)}{H(x\!:\ y)\ \textbf{corr}\ [P(x), Q(x, y)]}$$

Конкретизация правила:

```
      pNun(k, m, valD, res0, res) & m <> k+1 & qNun(k, m-1, valD, res0, res1) & digit(s[m-1], va) &
      res= res1*base+va -> qNun(k, m, valD, res0, res)
      pNun(k, m, valD, res0, res) & m <> k+1 & qNun(k, m-1, valD, res0, res1) & digit(s[m-1], va)
     ------------------------------------------------------------------------------------------------
COR: res= res1*base+va corr
      [pNun(k, m, valD, res0, res) & m <> k+1 & qNun(k, m-1, valD, res0, res1) & digit(s[m-1], va),
      qNun(k, m, valD, res0, res)]
```

Первая посылка определяет формулу корректности:

```
COR12: pNun(k, m, valD, res0, res) & m <> k+1 & qNun(k, m-1, valD, res0, res1) &
           digit(s[m-1], va) & res= res1*base+va -> qNun(k, m, valD, res0, res)
```

В итоге получим теорию:

```
formula pNun(k, m, valD, res0, res) =
      k < m /\ digit s[k] valD /\ numRes (k + 1) m ((res0 * base) + valD) res
formula qNun(k, m, valD, res0, res) = numRes k m res0 res
COR11: pNun(k, m, valD, res0, res) & m = k+1 ⇒ qNun(k, m, valD, res0, res);
QSB3: pNun(k, m, valD, res0, res) & m <> k+1 ->
            pNun(k, m-1, valD, res0, res1) & isDigit(s[m-1]);
COR12: pNun(k, m, valD, res0, res) & m <> k+1 & qNun(k, m-1, valD, res0, res1) &
           digit(s[m-1], va) & res= res1*base+va -> qNun(k, m, valD, res0, res)

predicate pNun(k m valD res0 res: int) =
      k < m /\ digit s[k] valD /\ numRes (k + 1) m ((res0 * base) + valD) res
predicate qNun(k m valD res0 res) = numRes k m res0 res
goal COR11: pNun k m valD res0 res  /\ m = k+1 -> qNun k m valD res0 res
goal QSB3: pNun k m valD res0 res  /\ m <> k+1 ->
            pNun k (m-1) valD res0 res1 /\ isDigit s[m-1]
goal COR12: pNun(k, m, valD, res0, res) & m <> k+1 & qNun(k, m-1, valD, res0, res1) /\
           digit s[m-1] va /\ res = ((res1*base)+va) -> qNun k m valD res0 res
```

B.2. Лемма **NuEq**.

При доказательстве возникла следующая лемма:

```
lemma NuEq :  forall k:int, m:int, res0:int, res:int, res1:int.
              numRes k m res0 res /\ numRes k m res0 res1 -> res1 = res
```

Построим следующую соответствующую программу в качестве леммы-предиката:

```
NuEq(int k, m, res, res0, res1)
     post numRes k m res0 res /\ numRes k m res0 res1 -> res1 = res;
     measure m
{   if (k = m) true  else  { NuEq(k, m-1, res, res0, res1)} }
```

```
formula pNuEq(int k, m, res, res0, res1) = true;
formula qNuEq(int k, m, res, res0, res1) =
        numRes k m res0 res /\ numRes k m res0 res1 -> res1 = res;
```

Введем формулу для меры.

```
formula hm(nat m: nat) = m;
```

Применим правило **QC**.

```
     B(x: y) corr [P(x) & E(x), Q(x, y)];
     C(x: z) corr [P(x) & ¬E(x), Q(x, y)]
QC: ----------------------------------------------
     {if (E(x)) B(x: y) else C(x: y)} corr [P(x), Q(x, y)]
```

Конкретизация правила **QC**:

```
    true corr [pNuEq(k, m, res, res0, res1) & k = m, qNuEq(k, m, res, res0, res1)];
    NuEq(k, m-1, res, res0, res1) corr
    [pNuEq(k, m, res, res0, res1) & k <> m, qNuEq(k, m, res, res0, res1)]
QC: ------------------------------------------------------------------------------
    numRes k m res0 res /\ numRes k m res0 res1 -> res1 = res corr
    [pNuEq(k, m, res, res0, res1), qNuEq(k, m, res, res0, res1)]
```

Для первой посылки применим правило **COR**.

```
      ∀x,y. P(x) & H(x: y) ⇒ Q(x, y);
      ∀x. P(x) ⇒ ∃y. H(x: y)
COR: ---------------------------------
      H(x: y) corr  [P(x), Q(x, y)]
```

Конкретизация правила:

```
     pNuEq(k, m, res, res0, res1) & k = m & true ⇒ qNuEq(k, m, res, res0, res1);
     pRnumR(js, valD, res) & ks = js ⇒ ∃b. true
COR: ---------------------------------------------------------------------------
     true corr [pNuEq(k, m, res, res0, res1) & k = m, qNuEq(k, m, res, res0, res1)]
```

Первая посылка определяет формулу корректности:

COR13: pNuEq(k, m, res, res0, res1) & k = m ⇒ qNuEq(k, m, res, res0, res1);

Вторая посылка правила **QC** определяет цель:

NuEq(k, m-1, res, res0, res1) **corr** [k <> m, qNuEq(k, m, res, res0, res1)]

Здесь учтено применение правила **QS**.

Применим правило **RB**.

```
     ∀z C(x, z: y) corr* [Pc(x, z), Qc(x, y)];
     P(x) ⇒ PB(x) & P*c(x, B(x));
     ∀y ( P(x) & Qc(B(x), y) ⇒ Q(x, y) );
RB: ----------------------------------------
       C(x, B(x): y) corr [P(x), Q(x, y)]
```

Конкретизации правила:

```
    RnumR(int js, valD, res) corr* [pRnumR(js-1, va, res), qRnumR(js-1, va, res)];
    k <> m ⇒ m-1<m;
    k <> m & qNuEq(k, m-1, res, res0, res1) ⇒ qNuEq(k, m, res, res0, res1)
RB: ------------------------------------------------------------------------------
    NuEq(k, m-1, res, res0, res1) corr [k <> m, qNuEq(k, m, res, res0, res1)]
```

Посылки правила **RB** определяют формулы корректности:

RB5: k <> m & qNuEq(k, m-1, res, res0, res1) ⇒ qNuEq(k, m, res, res0, res1)

В итоге получим теорию:

formula qNuEq(**int** k, m, res, res0, res1) =
    numRes k m res0 res /\ numRes k m res0 res1 -> res1 = res;
  COR13: k = m ⇒ qNuEq(k, m, res, res0, res1);
  RB5: k <> m & qNuEq(k, m-1, res, res0, res1) ⇒ qNuEq(k, m, res, res0, res1)

```
predicate qNuEq(k m res res0 res1: int) =
            numRes k m res0 res /\ numRes k m res0 res1 -> res1 = res
goal COR13: k = m -> qNuEq k m res res0 res1
goal RB5: k <> m /\ qNuEq k (m-1) res res0 res1 -> qNuEq k m res res0 res1
```