

A Visual Software Development Environment that Considers Tests of Physical Units *

Takaaki Goto¹, Yasunori Shiono², Tomoo Sumida², Tetsuro Nishino¹,
Takeo Yaku³ and Kensei Tsuchida²

¹*The University of Electro-Communications*

²*Toyo University*

³*Nihon University*
Japan

1. Introduction

Embedded systems are extensively used in various small devices, such as mobile phones, in transportation systems, such as those in cars or aircraft, and in large-scale distributed systems, such as cloud computing environments. We need a technology that can be used to develop low-cost, high-performance embedded systems. This technology would be useful for designing, testing, implementing, and evaluating embedded prototype systems by using a software simulator.

So far, embedded systems are typically used only in machine controls, but it seems that they will soon also have an information processing function. Recent embedded systems target not only industrial products but also consumer products, and this appears to be spreading across various fields. In the United States and Europe, there are large national projects related to the development of embedded systems. Embedded systems are increasing in size and becoming more complicated, so the development of methodologies and efficient testing for them is highly desirable.

The authors have been engaged in the development of a software development environment based on graph theory, which includes graph drawing theory and graph grammars [2–4]. In our research, we use Hichart, which is a program diagram methodology originally introduced by Yaku and Futatsugi [5].

There has been a substantial amount of research devoted to Hichart. A prototype formulation of attribute graph grammar for Hichart was reported in [6]. This grammar consists of Hichart syntax rules, which use a context-free graph grammar [7], and semantic rules for layout. The authors have been developing a software development environment based on graph theory that includes graph drawing theory and various graph grammars [2, 8]. So far, we have developed bidirectional translators that can translate a Pascal, C, or DXL source into Hichart and can alternatively translate Hichart into Pascal, C, or DXL [2, 8]. For example, HiChart Graph Grammar (HCGG) [9] is an attribute graph grammar with an underlying

*Part of the results have previously been reported by [1]

graph grammar based on edNCE graph grammar [10] and intended for use with DXL. It is problematic, however, in that it cannot parse very efficiently. Hichart Precedence Graph Grammar (HCPGG) was introduced in [11].

In recent years, model checking methodologies have been applied to embedded systems. In our current work, we constructed a visual software development environment to support a developed embedded system. The target of this research is NQC, which is the program language for LEGO MINDSTORM. Our visual software development system for embedded systems can

1. generate Promela codes for given Hichart diagrams, and
2. detect problems by using visual feedback features.

Our previously developed environment was not sufficiently functional, so we created an effective testing environment for the visual environment.

In this chapter, we describe our visual software development environment that supports the development of embedded systems.

2. Preliminaries

2.1 Embedded systems

An embedded system is a system that controls various components and specific functions of the industrial equipment or consumer electronic device it is built into [12, 13]. Product life cycles are currently being shortened, and the period from development to verification has now been trimmed down to about three months. Four requirements are needed to implement modern embedded systems.

- **Concurrency**
Multi-core and/or multi processors are becoming dominant in the architecture of processors as a solution to the limits in circuit line width (manufacturing process), increased generation of heat, and clock speed limits. Therefore, it is necessary to implement applications by using methods with parallelism descriptions.
- **Hierarchy**
System modules are arranged in a hierarchal fashion in main systems, subsystems, and sub-subsystems. Diversity and recycling must be improved, and the number of development processes should be reduced as much as possible.
- **Resource Constraints**
It is necessary to comply with the constraints of built-in factors like memory and power consumption.
- **Safety and Reliability**
System failure is a serious problem that can cause severe damage and potentially fatal accidents. It is extremely important to guarantee the safety of a system.

LEGO MINDSTORMS [14] is a robotics environment that was jointly developed by the REGO and MIT. MINDSTORMS consists of a block with an RCX or NXT micro processor. Robots that are constructed with RCX or NXT and sensors can work autonomously, so a block with RCX or NXT can control a robot's behavior. RCX or NXT detects environment information through

attached sensors and then activates motors in accordance with the programs. RCX and NXT are micro processors with a touch sensor, humidity sensor, photodetector, motor, and lamp.

ROBOLAB is a programming environment developed by National Instruments, the REGO, and Tufts University. It is based on LABVIEW (developed by National Instruments) and provides a graphical programming environment that uses icons.

It is easy for users to develop programs in a short amount of time because ROBOLAB uses templates. These templates include various icons that correspond to different functions which then appear in the developed program in pilot level. ROBOLAB has fewer options than LABVIEW, but it does have some additional commands that have been customized for RCX.

Two programming levels, pilot level and inventor level, can be used in ROBOLAB. The steps then taken to construct a program are as follows.

1. Choose icons from palette.
2. Put icons in a program window.
3. Set orders of icons and then connect them.
4. Transfer obtained program to the RCX.

Not Quite C (NQC) [15] is a language that can be used in LEGO MINDSTORM RCX. Its specification is similar to that of C language, but differs in that it does not provide a pointer but instead has functions specialized for LEGO MINDSTORMS, including "turn on motors," "check touch sensors value," and so on.

A typical NQC program starts from a "main" task and can handle a maximum of ten tasks. When we write NQC source codes, the below description is required.

Listing 1. Example1

```
task main()
{
}
```

Here, we investigate functions and constants. The below program shows MINDSTORMS going forward for four seconds, then backward for four seconds, and then stopping.

Listing 2. Example2

```
task main()
{
    OnFwd(OUT_A+OUT_C);
    Wait(400);
    OnRev(OUT_A+OUT_C);
    Wait(400);
    Off(OUT_A+OUT_C);
}
```

Here, the functions "OnFwd," "OnRev," etc. control RCX. Table 1 shows an example of functions customized for NQC.

Functions	Explanation	Example of description
SetSensor(<sensor name>, <configuration>)	set type and mode of sensors	SetSensor(SENSOR_1, SENSOR_TOUCH)
SetSensorMode(<sensor name>, <mode>)	set a sensor's mode	SetSensorMode(SENSOR_2, SENSOR_MODE_PERCENT)
OnFwd(<outputs>)	set direction and turn on	OnFwd(OUT_A)

Table 1. Functions of RCX

As for the constants, they are constants with names and work to improve programmers' understanding of NQC programs.

Table 2 shows an example of constants.

Constants category	Constants
Setting for SetSensor()	SENSOR_MODE_RAW, SENSOR_MODE_BOOL, SENSOR_MODE_EDGE, SENSOR_MODE_PULSE, SENSOR_MODE_PERCENT, SENSOR_MODE_CELCIUS, SENSOR_MODE_FAHRENHEIT, SENSOR_MODE_ROTATION
Mode for SetSensorMode	SENSOR_MODE_RAW, SENSOR_MODE_BOOL, SENSOR_MODE_EDGE, SENSOR_MODE_PULSE, SENSOR_MODE_PERCENT, SENSOR_MODE_CELCIUS, SENSOR_MODE_FAHRENHEIT, SENSOR_MODE_ROTATION

Table 2. Constants of RCX

We adopt REGO MINDSTORMS as an example of embedded systems with sensors.

2.2 Program diagrams

In software design and development, program diagrams are often used for software visualization. Many kinds of program diagrams, such as the previously mentioned hierarchical flowchart language (Hichart), problem analysis diagram (PAD), hierarchical and compact description chart (HCP), and structured programming diagram (SPD), have been used in software development [2, 16]. Moreover, software development using these program diagrams is steadily on the increase.

In our research, we used the Hichart program diagram [17], which was first introduced by Yaku and Futatsugi [5]. Figure 1 shows a program called "Tower of Hanoi" that was written in Hichart.

Hichart has three key features:

1. A tree-flowchart diagram that has the flow control lines of a Neumann program flowchart,

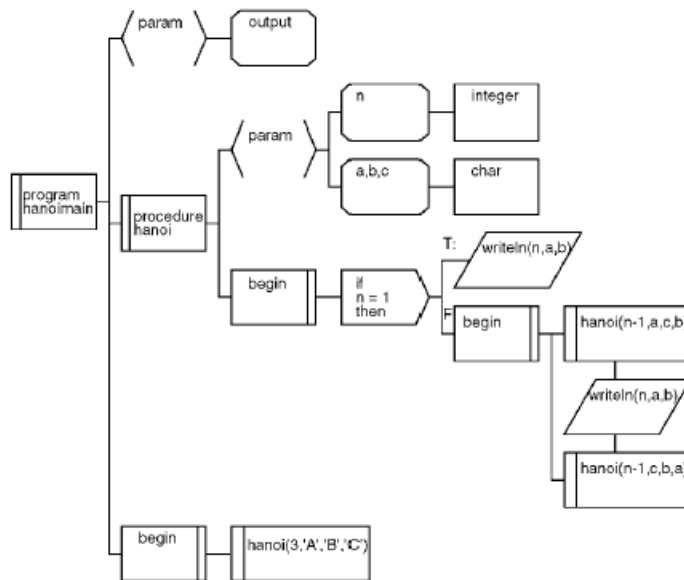


Fig. 1. Example of Hichart: “Tower of Hanoi”.

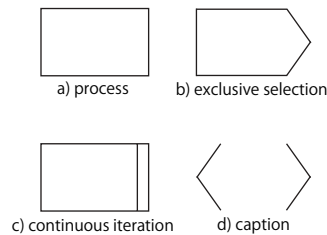


Fig. 2. Example of Hichart symbols.

2. Nodes of the different functions in a diagram that are represented by differently shaped cells, and
3. A data structure hierarchy (represented by a diagram) and a control flow that are simultaneously displayed on a plane, which distinguishes it from other program diagram methodologies.

Hichart is described by cell and line. There are various type of cells, such as "process," "exclusive selection," "continuous iteration," "caption," and so on. Figure 2 shows an example of some of the Hichart symbols.

3. Program diagrams for embedded systems

In this section, we describe program diagrams for embedded systems, specifically, a detailed procedure for constructing program diagrams for an embedded system using Hichart for NQC.

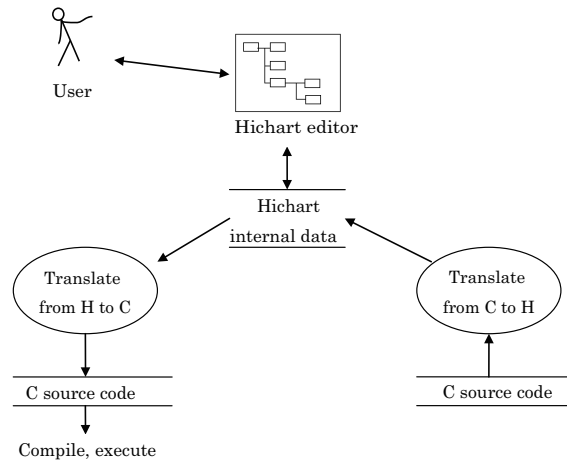


Fig. 3. Overview of our previous study.

Figure 3 shows an overview of our previous study on a Hichart-C translation system.

In our previous system, it is possible to obtain internal Hichart data from C source code via a C-to-H translator implemented using JavaCC. Users can edit a Hichart diagram on a Hichart editor that visualizes the internal Hichart data as a Hichart diagram. The H-to-C translator can generate C source codes from the internal Hichart data, and then we can obtain the C source code corresponding to the Hichart diagrams. Our system can illustrate programs as diagrams, which leads to an improved understanding of programs.

We expanded the above framework to treat embedded system programming. Specifically we extended H-to-C and C-to-H specialized for NQC. Some of the alterations we made are as follows.

1. task
The “task” is a unique keyword of NQC, and we therefore added it to the C-to-H function.
2. start, stop
We added “start” and “stop” statements in Hichart (as shown in List 3) to control tasks.

Listing 3. Example3

```

task main()
{
    SetSensor (SENSOR_1,SENSOR_TOUCH);
    start check_sensors;
    start move_square;
}
task move_square()
{
    while(true)
    {
        OnFwd(OUT_A+OUT_C); Wait(100);
    }
}
  
```

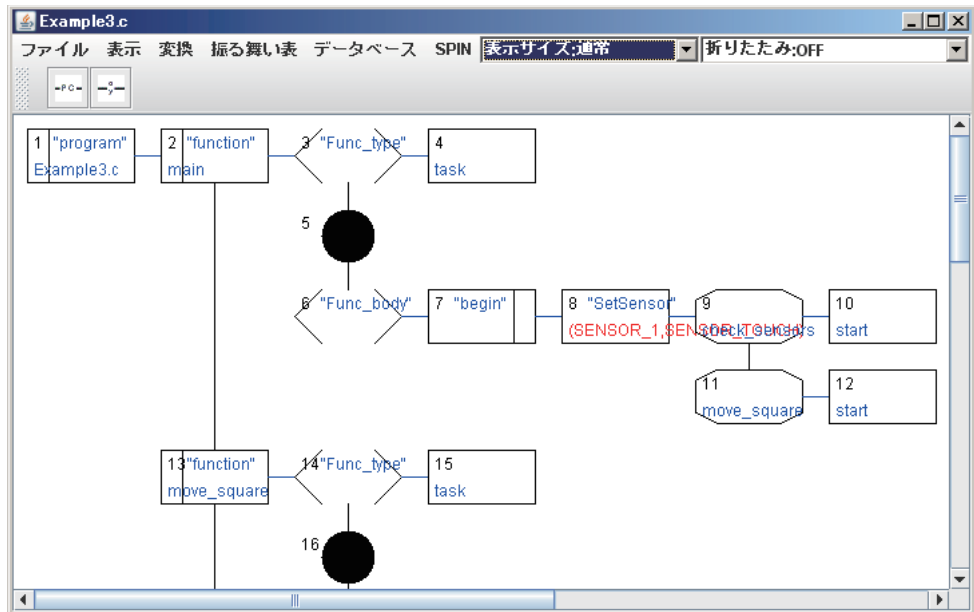


Fig. 4. Screenshot of Hichart for NQC that correspond to List 3.

```

    OnRev(OUT_C); Wait(68);
}
}

task check_sensors()
{
    while(true)
    {
        if (SENSOR_1 == 1)
        {
            stop move_square;
            OnRev(OUT_A+OUT_C); Wait(50);
            OnFwd(OUT_A); Wait(85);
            start move_square;
        }
    }
}

```

There are some differences between C syntax and NQC syntax; therefore, we modified JavaCC, which defines syntax, to cover them. Thus, we obtained program diagrams for embedded systems.

Figure 4 shows a screenshot of Hichart for NQC that correspond to List 3.

4. A visual software development environment

We propose a visual software development environment based on Hichart for NQC. We visualize NQC code by the abovementioned Hichart diagrams through a Hichart visual software development environment called Hichart editor. Hichart diagrams or NQC source codes are inputted into the editor, and the editor outputs NQC source codes after editing code such as parameter values in diagrams.

In the Hichart editor, the program code is shown as a diagram. List 4 shows a sample program of NQC, and Figure 5 shows the Hichart diagram corresponding to List 4.

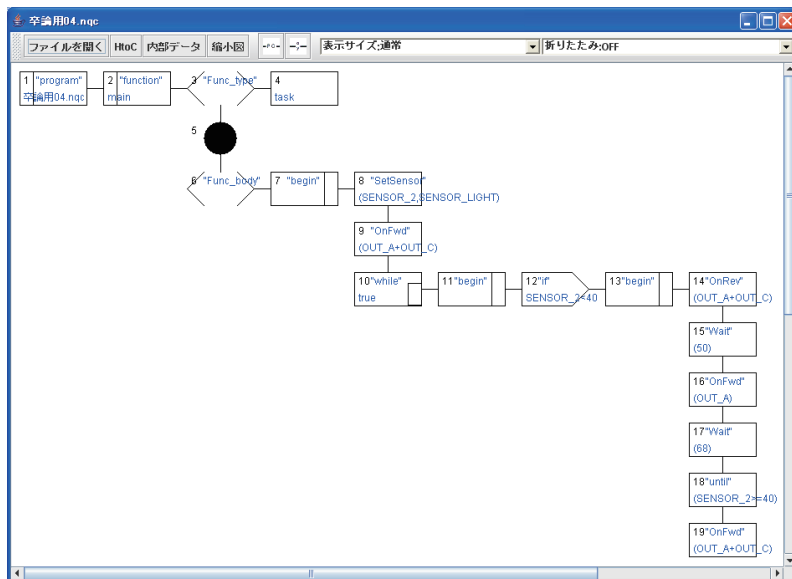


Fig. 5. Screen of Hichart editor.

Listing 4. anti-drop program

```
task main ()
{
  SetSensor (SENSOR_2,SENSOR_LIGHT);
  OnFwd(OUT_A+OUT_C);
  while( true )
  {
    if (SENSOR_2 < 40)
    {
      OnRev (OUT_A+OUT_C);
      Wait (50);
      OnFwd (OUT_A);
      Wait (68);
      until (SENSOR_2 >= 40);
      OnFwd (OUT_A+OUT_C);
    }
  }
}
```



```

    }
  }
}

```

This Hichart editor for NQC has the following characteristics.

1. Generation of Hichart diagram corresponding to NQC
2. Editing of Hichart diagrams
3. Generation of NQC source codes from Hichart diagrams
4. Layout modification of Hichart diagrams

Users can edit each diagram directly on the editor. For example, cells can be added by double-clicking on the editor screen, after which cell information, such as type and label, is embedded into the new cell.

Figure 6 shows the Hichart screen after diagram editing. In this case, some of the parameter's values have been changed.

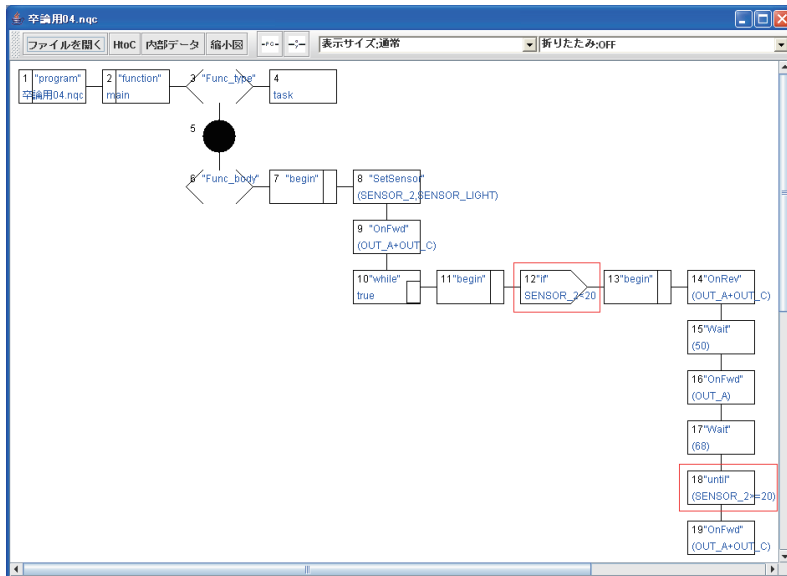
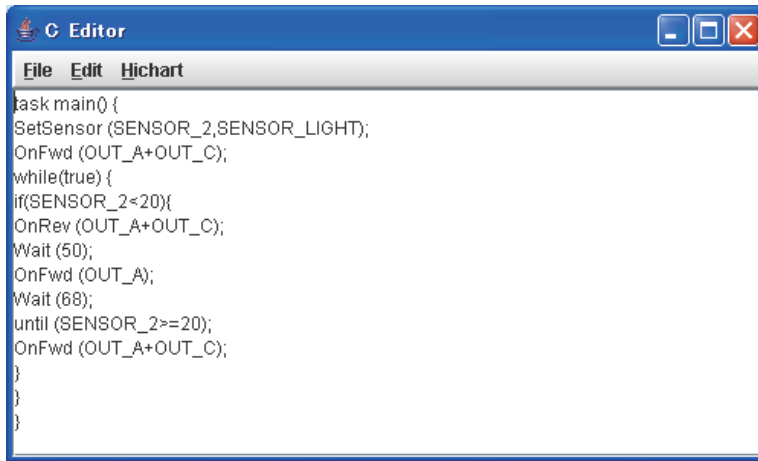


Fig. 6. Hichart editor screen after editing.

The Hichart editor can read NQC source codes and convert them into Hichart codes using the N-to-H function, and it can generate NQC source codes from Hichart codes by using the H-to-N function. The Hichart codes consist of tree data structure. Each node of the structure has four pointers (to parent node, to child cell, to previous cell, and to next cell) and node information such as node type, node label, node label, and so on. To generate NQC codes by the H-to-N function, tree structures can be traversed in preorder.

The obtained NQC source code can be transferred to the LEGO MINDSTORM RCX via BricxCC. Figure 7 shows a screenshot of NQC source code generated by the Hichart editor.



```

task main() {
  SetSensor (SENSOR_2,SENSOR_LIGHT);
  OnFwd (OUT_A+OUT_C);
  while(true) {
    if(SENSOR_2<20){
      OnRev (OUT_A+OUT_C);
      Wait (50);
      OnFwd (OUT_A);
      Wait (68);
    }
    until (SENSOR_2>=20);
    OnFwd (OUT_A+OUT_C);
  }
}

```

Fig. 7. Screenshot of NQC source code generated by Hichart editor.

Sensitivity s	0-32	33-49	50-100
Recognize a table edge	×	○	○
Turn in its tracks	○	○	×

Table 3. Behavioral specifications table.

5. Testing environment based on behavioral specification and logical checking

To test embedded system behaviors, especially for those that have physical devices such as sensors, two areas must be checked: the value of the sensors and the logical correctness of the embedded system. Embedded systems with sensors are affected by the environment around the machine, so it is important that developers are able to set the appropriate sensor value. Of course, even if the physical parameters are appropriate, if there are logical errors in a machine's program, the embedded systems will not always work as we expect.

In this section, we propose two testing methods to check the behaviors of embedded systems.

5.1 Behavioral specifications table

A behavioral specifications table is used when users set the physical parameters of RCX. An example of such a table is shown in Table 3. The leftmost column lists the behavioral specifications and the three columns on the right show the parameter values. A circle indicates an expected performance; a cross indicates an unexpected one. The numerical values indicate the range of sensitivity parameters s .

For example, when the sensitivity parameter s was between 0 and 32, the moving object did not recognize a table edge (the specifications for "recognizes a table edge" were not met) and did not spin around on that spot. When the sensitivity parameter s was between 33 and 49, the specifications for "recognizes a table edge" and "does not spin around on that spot" were both met.

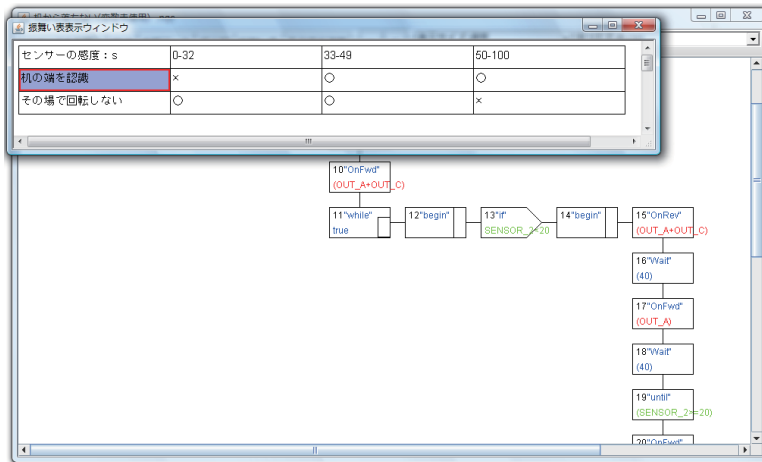


Fig. 8. Screenshot of Hichart editor and behavioral specifications table.

The results in the table show that the RCX with a sensor value from 0 to 32 cannot distinguish the edge of the table and so falls off. Therefore, users need to change the sensor value to the optimum value by referencing the table and choosing the appropriate value. In this case, if users only choose the column with the values from 33 to 49, the chosen value is reflected in the Hichart diagram. This modified Hichart diagram can then generate an NQC source code. This is an example of how developers can easily set appropriate physical parameters by using behavioral specifications tables.

The behavioral specifications function has the following characteristics.

1. The editor changes the colors of Hichart cells that are associated with the parameters in the behavioral specifications table.
2. The editor sets the parameter value of Hichart cells that are associated with the parameters in the behavioral specifications table.

Here, we show an example in which an RCX runs without falling off a desk. In this example, when a photodetector on the RCX recognizes the edge of the desk, the RCX reverses and turns. Figure 8 shows a screenshot of the Hichart editor and the related behavioral specifications table.

In the Hichart editor, the input-output cells related to a behavioral specifications table are redrawn in green when the user chooses a menu that displays the behavioral specifications table.

Figure 9 shows the behavior of an RCX after setting the appropriate physical parameters. The RCX can distinguish the table edge and turn after reversing.

We also constructed a function that enables a behavioral specification table to be stored in a database that was made using MySQL. After we test a given device, we can input the results via the database function in the Hichart editor. Using stored information, we can construct a behavioral specification table with an optimized parameter's value.

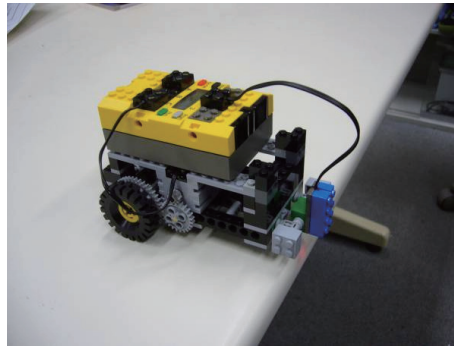


Fig. 9. Screenshot of RCX that recognizes table edge.

5.2 Model checking

We propose a method for checking behavior in the Hichart development environment by using the model checking tool SPIN [18, 19] to logically check whether a given behavior specification is fulfilled before applying the program to a real machine. As described previously, the behavioral specifications table can check the physical parameters of a real machine. However, it cannot check logical behavior. We therefore built a model checking function into our editor that can translate internal Hichart data into Promela code.

The major characteristics of the behavior specification verification function are listed below.

- Generation of Promela codes
Generating Promela codes from Hichart diagrams displayed on the Hichart editor.
- Execution of SPIN
Generating pan.c or LTL-formulas.
- Compilation
Compiling obtained pan.c to generate .exe file for model checking.
- Analyzing
- Analysis
We found that programs do not bear the behavior specification by model checking and so generated trail files. The function then analyzes the trail files and feeds them back to the Hichart diagrams.

The Promela code is used to check whether a given behavior specification is fulfilled. Feedback from the checks is then sent to a Hichart graphical editor. If a given behavioral specification is not fulfilled, the result of the checking is reflected in the implicated location of the Hichart.

To give an actual example, we consider the specifications that make the RCX repeat forward movements and turn left. If it is touch sensitive, the RCX changes course. This specification means that RCX definitely swerves when touched. In this study, we checked whether the created program met the behavior specification by using SPIN before applying the program to real machines.

Listing 5. Source code of NQC

```
task move_square(){
  while(true){
    OnFwd(OUT_A + OUT_C);
    Wait(1000);
    OnRev(OUT_C);
    Wait(85);
  }
}
```

Listing 6. Promela code

```
proctype move_square(){
  do
    ::
      state = OnFwd;
      state = Wait;
      state = OnRev;
      state = Wait;
  od
}
```

Lists 5 and 6 show part of the NQC source code corresponding to the above specification and the automatically generated Promela source code.

We explain the feedback procedure, which is shown in Fig. 10.

An assertion statement of “state == OnFwd” is an example. If a moving object (RCX) is moving forward at the point where the assertion is set, the statement is true. Otherwise, it is false. For example, we can verify by steps (3)-(7) in Fig. 10 whether the moving object is always moving forward or not.

Here, we show an example of manipulating our Hichart editor. We can embed an assertion description through the Hichart editor, as shown in Fig. 11, and then obtain a Promela code from the Hichart code. When we obtain this code, we have to specify the behaviors that we want to check. Figure 12 shows a result obtained through this process.

Next, we execute SPIN. If we embed assertions in the Hichart code, we execute SPIN as it currently stands, while if we use LTL-formulas, we execute SPIN with an “-f” option and then obtain pan.c. The model is checked by compiling the obtained pan.c. Figure 13 is a screenshot of the model checking result using the Hichart editor.

If there are any factors that do not meet the behavioral specifications, trail files are generated. Figure 14 shows some of the result of analyzing the trail file.

The trail files contain information on how frequently the processing calls and execution paths were made. We use this information to narrow the search area of the entire program by using the visual feedback. Users can detect a problematic area interactively by using the Hichart editor with the help of this visual feedback.

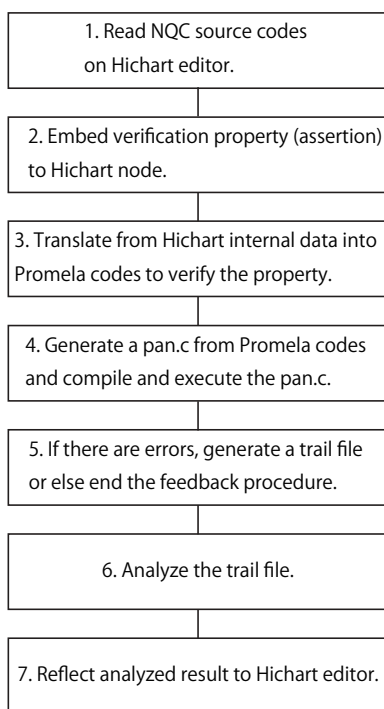


Fig. 10. Feedback procedure.

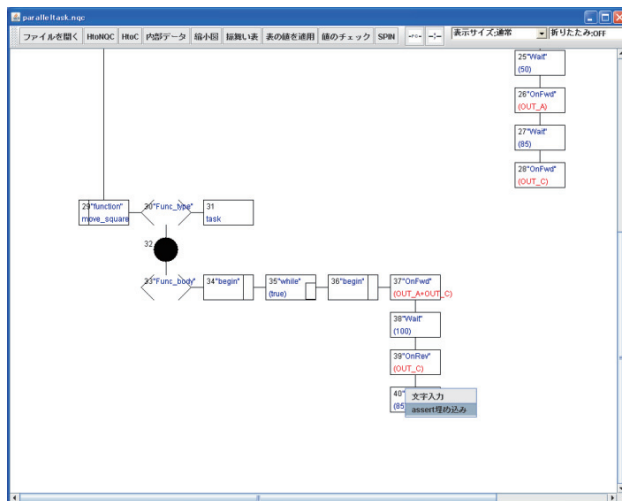


Fig. 11. Embed an assertion on Hichart editor.

```
fi
od
]

/* end */

active proctype main() {
run check_sensors0;
run move_square0;
}

proctype check_sensors0 {
do
::
if
:: SENSOR == on ->
state = OnRev;
move(state, place);
state = Wait;
touch(place);
state = OnFwd;
move(state, place);
:: else -> skip
fi
od
}
```

Fig. 12. Result of generating a Promela code.

```
pan: assertion violated (state==6) (at depth 63)
pan: wrote paralletask_assert.p.trail

(Spin Version 5.1.6 -- 9 May 2008)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
never claim - (none specified)
assertion violations +
acceptance cycles - (not selected)
invalid end states +

State-vector 28 byte, depth reached 63, errors: 1
64 states, stored
21 states, matched
85 transitions (= stored+matched)
0 atomic steps
hash conflicts: 0 (resolved)

2.501 memory usage (Mbyte)

pan: elapsed time 0 seconds
```

Fig. 13. Result of model checking.

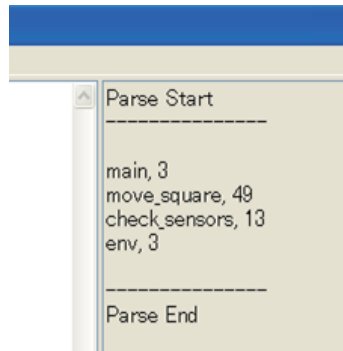


Fig. 14. Result of analyzing trail file.

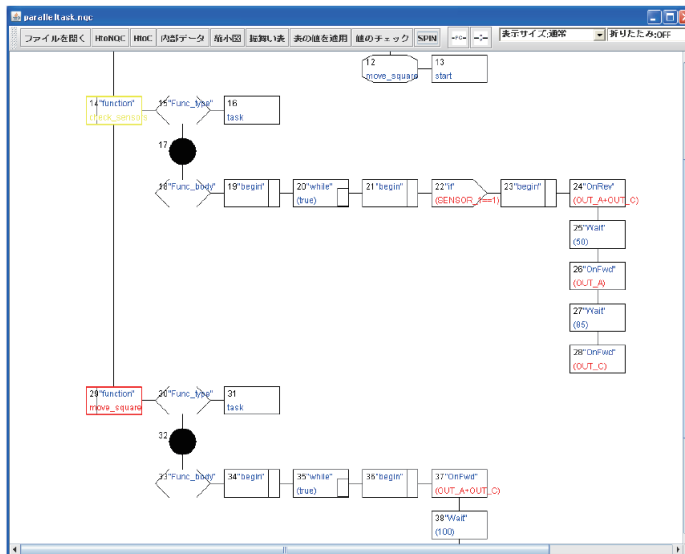


Fig. 15. Part of Hichart editor feedback screen.

After analyzing the trail files, we can obtain feedback from the Hichart editor. Figure 15 shows part of a Hichart editor feedback screen.

If the result is that programs did not meet the behavior specification by using SPIN, the tasks indicated as the causes are highlighted. The locations that do not meet the behavior specifications can be seen by using the Hichart feedback feature. This is an example of efficient assistance for embedded software.

6. Conclusion

We described our application of a behavioral specification table and model-checking methodologies to a visual software development environment we developed for embedded software.

A key element of our study was the separation of logical and physical behavioral specifications. It is difficult to verify behaviors such as those of robot sensors without access to the behaviors of real machines, and it is also difficult to simulate behaviors accurately. Therefore, we developed behavioral specification tables, a model-checking function, and a method of giving visual feedback.

It is rather difficult to set exact values for physical parameters under development circumstances using a tool such as MATLAB/simulink because the physical parameters vary depending on external conditions (e.g., weather), and therefore, there were certain limitations to the simulations. We obtained a couple of examples demonstrating the validity of our approach in both the behavioral specification table and the logical specification check by using SPIN.

In our previous work, some visual software development environments were developed based on graph grammar; however, the environment for embedded systems described in this article is not yet based on graph grammars. A graph grammar for Hichart that supports NQC is currently under development.

In our future work, we will construct a Hichart development environment with additional functions that further support the development of embedded systems.

7. References

- [1] T. Goto, Y. Shiono, T. Nishino, T. Yaku, and K. Tsuchida. Behavioral verification in hichart development environment for embedded software. In *Computer and Information Science (ICIS), 2010 IEEE/ACIS 9th International Conference on*, pages 337–340, aug. 2010.
- [2] K. Sugita, A. Adachi, Y. Miyadera, K. Tsuchida, and T. Yaku. A visual programming environment based on graph grammars and tidy graph drawing. In *Proceedings of The 20th International Conference on Software Engineering (ICSE '98)*, volume 2, pages 74–79, 1998.
- [3] T. Goto, T. Kirishima, N. Motousu, K. Tsuchida, and T. Yaku. A visual software development environment based on graph grammars. In *Proc. IASTED Software Engineering 2004*, pages 620–625, 2004.
- [4] Takaaki Goto, Kenji Ruise, Takeo Yaku, and Kensei Tsuchida. Visual software development environment based on graph grammars. *IEICE transactions on information and systems*, 92(3):401–412, 2009.
- [5] Takeo Yaku and Kokichi Futatsugi. Tree structured flow-chart. In *Memoir of IEICE*, pages AL-78, 1978.
- [6] T. Nishino. Attribute graph grammars with applications to hichart program chart editors. In *Advances in Software Science and Technology*, volume 1, pages 89–104, 1989.
- [7] C. Ghezzi P. D. Vigna. Context-free graph grammars. In *Information Control*, volume 37, pages 207–233, 1978.
- [8] Y. Adachi, K. Anzai, K. Tsuchida, and T. Yaku. Hierarchical program diagram editor based on attribute graph grammar. In *Proc. COMPSAC*, volume 20, pages 205–213, 1996.
- [9] Masahiro Miyazaki, Kenji Ruise, Kensei Tsuchida, and Takeo Yaku. An NCE Attribute Graph Grammar for Program Diagrams with Respect to Drawing Problems. *IEICE Technical Report*, 100(52):1–8, 2000.

- [10] Grzegorz Rozenberg. *Handbook of Graph Grammar and Computing by Graph Transformation Volume 1*. World Scientific Publishing, 1997.
- [11] K. Ruise, K. Tsuchida, and T. Yaku. Parsing of program diagrams with attribute precedence graph grammar. In *Technical Report of IPSJ*, number 27, pages 17–20, 2001.
- [12] R. Zurawski. *Embedded systems design and verification*. CRC Press, 2009.
- [13] S. Narayan. Requirements for specification of embedded systems. In *ASIC Conference and Exhibit, 1996. Proceedings., Ninth Annual IEEE International*, pages 133–137, sep 1996.
- [14] LEGO. LEGO mindstorms. <http://mindstorms.lego.com/en-us/Default.aspx>.
- [15] Not Quite C. <http://bricxcc.sourceforge.net/nqc/>.
- [16] Kenichi Harada. *Structure Editor*. Kyoritsu Shuppan, 1987. (in Japanese).
- [17] T. Yaku, K. Futatsugi, A. Adachi, and E. Moriya. HICHART -A hierarchical flowchart description language-. In *Proc. IEEE COMPSAC*, volume 11, pages 157–163, 1987.
- [18] G.J. Holzmann. The model checker spin. *Software Engineering, IEEE Transactions on*, 23(5):279–295, may 1997.
- [19] M. Ben-Ari. *Principles of the SPIN Model Checker*. Springer, 2008.