Proceedings of the 2001 IEEE
International Conference on Robotics & Automation
Seoul, Korea · May 21-26, 2001

Creating the Architecture of a Translator Framework for Robot Programming Languages

Eckhard Freund, Bernd Lüdemann-Ravit, Oliver Stern, Thorsten Koch

Institute of Robotics Research (IRF), University of Dortmund
Otto-Hahn-Str. 8
D-44227 Dortmund, Germany
email: {freund,luedeman,stern,koch}@irf.de

Abstract

This paper presents a novel approach to facilitate the development and maintenance of translators for industrial robot programming languages. Such translators are widely used in robot simulation and offline programming systems to support programming in the respective native robot language. Our method is based upon a software architecture, that is provided as a complete translator framework. For the developer of a new translator, it offers convenient strategies to concentrate on robot specific language elements during the design and implementation process: fill-in *templates*, *libraries* for common functionality, design patterns etc., all tied up with a general translation scheme. In contrast to other compiler construction tools, the developers need not care about the complex details of a whole translator. As a matter of principle, the architecture offers a complete default translator (except for the grammar). Robot specific elements can be held in separate units - outside of the actual translator - to facilitate maintenance and feature extension. The most probable changes in the translator product life cycle are restricted to the adaptation of these units. Several translators built upon this framework are in actual use in the commercial robot simulation system COSIMIR® to support native language robot programming, as well as in the widely used robot programming system COSIROP to verify the syntax of robot programs.

1 Introduction

There exist two different basic approaches to develop industrial robot programs within a robot simulation system. The first one uses a general engineering language which is translated into robot specific code for execution on robot controllers (post-processors). The second technique is characterized by programming the robot in its native robot programming language inside the simulation system [1].

Modern robot simulation systems should support native robot programming of industrial robots for two main reasons. First, robot programs executed in industrial environments are characterized by frequent changes, e.g. due to product or production cell changes. These changes are most often carried out directly on the shop floor. Simulation systems which support native robot programming can process these modified programs for further analysis, simulation or reprogramming. Secondly, the acceptance of a robot simulation system increases with the support for native robot programming. The users need not learn a new programming language, but can program in the robot language he is familiar with.

The main difficulty for robot simulation systems to support native robot programming is the fact that there exist several hundred different proprietary robot programming languages [2]. Each robot manufacturer has developed his own language. The robot programming language does even vary between different version of a robot controller. Some manufacturers support even more than one programming language with the same controller, e.g. Mitsubishi Electric [3].

Another problem is the history of the language design. The manufactures have changed, extended and modified their languages in the course of time. As a result, these languages are difficult to handle, because there is no strict systematics (e.g. lack of grammars) within the languages. Besides, modern production cells are very heterogeneous. Industrial robots from different manufactures and robot controllers with different software versions are found within the same workcell.

Consequently, the effort to support more than one language within a robot simulation or programming system is extremely high, and the development process is very expensive. In this paper, we present a framework to minimize the time-to-market for the support of new industrial robot programming languages within a robot simulation system, allowing the implementation of new translators on customer demand.

2 Requirements

To support different robot programming languages in a simulation system, all these languages are translated into the same "assembler-like" intermediate code, e.g. IRDATA [4] or ICR [5]. This code is interpreted by a virtual robot controller inside the simulation system. This method is a modified version of the proposal in [1] (see figure 1).

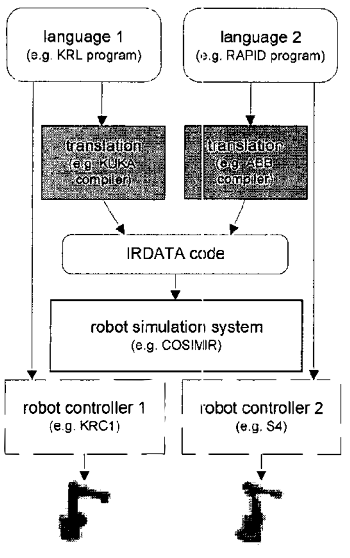

Figure 1: Native language support

Our experience has shown that the development time for such translators varies by about 30% (see table 1), depending on the complexity of the language. The IRL (Industrial Robot Language) translator was used as the reference for the values in the table.

Language (Manufacturer)	*Development time*
BAPS (Bosch)	80%
IRL (Standard)	100%
MBA3 (Mitsubishi)	90%
MRL (Mitsubishi)	70%
VAL II (Unimation)	100%

Table 1: Previous development times (in % of IRL)

Therefore, a software architecture is needed to support the programmer in the development process. The aim of our translation framework is to reduce the development and maintenance time.

We have identified the following main requirements for such an architecture:

(A) Common functionality for different languages should be unified and standardized in libraries.

(B) Language specific issues should be completed by filling in templates rather than programmed anew.

(C) Common functionality must be adaptable to the needs of language specific issues.

(D) It should be possible to use libraries of native and intermediate code to facilitate maintenance and to reduce the necessary amount of code for the translator.

So, the developer of a translator for a specific robot programming language should be put into the position to concentrate on the language specific issues and to neglect the implementation of recurring mechanisms.

3 System Structure

Our framework implements a complete translator for a general (higher) programming language, except for the grammar. The overall structure of the system consists of five main units, which correspond to the classic phases of compilation (see figure 2).

This concept is based on an advanced paradigm of translation: the translation steps are executed sequentially on an internal representation (abstract syntax tree, AST) of the complete robot program. In older compilers, each translation step was executed only on a single command. This means that only the necessary information to translate one command was held in memory [6], resulting in reduced memory consumption. The advantage of the newer concept is in the design of the translation system: the translation steps can be separated much more easily, thus operating independently on the AST [7]. This concept is therefore better suited to serve as a basis for the requirements shown above.

All robot programming languages have built-in elements, such as predefined variables (e.g. actual position), data types (e.g. positions), functions (e.g. frame manipulation) and statements (e.g. movement commands). These built-in elements distinguish a robot programming language from a "general" programming language. Some built-in elements are explicitly defined in the so called "system files" of the corresponding robot controller. These files are usually coded in the robot programming language.

The declarations of built-in elements, which cannot be found inside the system files, should be put into the built-in language elements files.

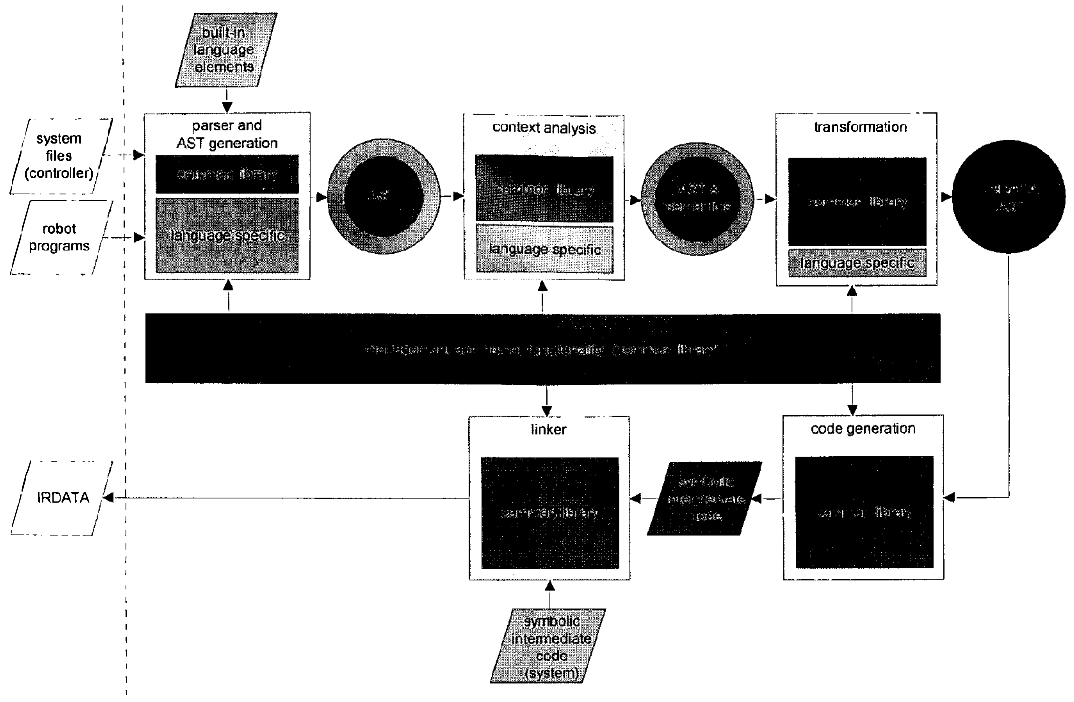

Figure 2: Structure of the translator framework

The translator can then use this information to perform a complete syntax and context analysis, and to generate a standardized call into the intermediate system code library, where the functionality has to be implemented in symbolic intermediate code.

The advantage of this approach is the fact, that for changing the semantics and/or implementation of one of these built-in elements, the translator itself need not be touched. Thus, considerable time in the test, correction and feature extension phases of the translator life cycle is saved. This facilitates maintenance enormously (see requirement (D)).

Due to the development history of some industrial robot programming languages, it is not always possible to formalize the declaration of built-in elements in one of these files. Such built-in elements must be added to the language specific AST, because they need a specific treatment inside the translator. The following example, a palletizing function, is taken from the Mitsubishi MELFA BASIC IV (MBA4) language [7]. The parameters are not allowed to be in brackets, which would be necessary for a declaration in ordinary syntax.

```
10  P1 = PLT 3,5
```

3.1 Design of the System Units

The main characteristic of this architecture is the division of each system unit and each data structure of the AST into a common part and a language specific part. In the common part, everything is united that can be found in "ordinary" programming languages (e.g. program flow instructions). A default processing for this part of the AST is offered to the developer of a new translator. In the language specific part, everything is concentrated that belongs to a specific robot language, and that cannot be declared in the built-in elements or system files.

Every portion of the AST is divided into a common library part and a language specific part in this manner. The division was made analogously for declarations, statements, expressions, actual and formal parameter, routines, etc. The following example illustrates the division of the statements with an indirect call (common library) and a linear movement (language specific part). We use GENTLE [7] notation, the PROLOG like programming language that was used to implement the framework.

```
-- common part
   'action' AnalyzeStmts(STMT -> STMT)
(I)      'rule' AnalyzeStmts(Stmt -> A_Stmt):
             CS_Pre_AnalyzeStmt(Stmt -> A_Stmt)
(II)     'rule' AnalyzeStmts(cc_callindir(Pos, ...) -> cc_callindir(Pos, ...):
(III)    'rule' AnalyzeStmts(cs_stmt(Pos, CsStmt) -> cs_stmt(Pos, A_CsStmt)):
             CS_AnalyzeStmt(CsStmt -> A_CsStmt)

-- language specific part
(i)    'action' CS_Pre_AnalyzeStmt(STMT -> STMT)

(ii)   'action' CS_AnalyzeStmt(CS_STMT -> CS_STMT)
         'rule' CS_AnalyzeStmt(mba4_mvs(Pos, ...) -> mba4_mvs(Pos, ...)):
         ...
```

Figure 3: Context analysis example

The common AST part comprises the following elements:

```
'type' STMT
  -- common part
  cc_callindir(Pos:    POS,
               Expr:   EXPR,
               AParam: APARAMLIST)
  -- language specific statements
  cs_stmt(Pos:  POS,
          Stmt: CS_STMT)
```

The language specific AST part includes the following:

```
'type' CS_STMT
  -- linear movement
  mba4_mvs(Pos:    POS,
           Target: EXPR,
           ...)
```

The common part includes one predicate (here, "cs_stmt") where all language specific statements can be included. So, common units can work on the common AST, while language specific units can process the language specific part. Thus, no common unit can see any language specific parts of the AST.

Every processing step ("action") on the AST in a common unit is structured in the same manner, see figure 3 for an example from the context analysis:

(I) One predicate to overwrite the standard processing of common AST parts, if desired (requirement (C)).

(II) Several common predicates that define a default behavior for the common AST processing (requirement (A)).

(III) One predicate to handle all language specific AST parts (requirement (B)).

For (I) and (III), there are corresponding processing actions (i) and (ii) in the language specific unit.

Action (i) is filled-in in the language specific unit, if the processing of the common AST is insufficiently covered by the common rules. Usually, this action is empty, so the default processing is used. Thus, it is possible to add some processing steps to the common rules from (II), that are carried out before the default behavior. But also, the complete default processing can be overwritten.

Action (ii) must be implemented for all language specific statements, that exist inside the language specific AST, and that are not declared in the built-in elements or system files (e.g. "mba4_mvs()").

3.2 Recipe for new Translators

To develop a new translator for a -- maybe even exotic - robot programming language, the following main steps have to be performed:

1. Define a grammar. This has to be done in any case. Except for expressions, there is no common grammar part available. For expressions, design patterns from the common library can be used. Due to the compiler construction kit used here [7], this step is well supported.

2. Determine the built-in elements of the language (check the manuals).

   2.1. Analyze the controller's system files for declarations of built-in elements.

2.2. If other built-in elements can be declared in the native robot language, declare them in the built-in elements files.

2.3. Add the remaining built-in elements to the language specific AST.

3. Fill in the templates in the language specific units (context analysis, transformation) of the framework to handle the parts of the language specific AST. (ii)

4. Add language specific handling of built-in data types inside expressions to the context analysis unit. (ii)

5. Add type compatibility and conversion issues to the context analysis unit, especially for assignment, actual and formal parameters, and function return values. (ii)

6. Analyze the language for elements, that are treated in another way than in the common part, and override the corresponding behavior inside the language specific system units. (i)

7. Implement built-in statements and functions inside the symbolic intermediate system code file.

4 Results

In table 2, 3 and 4, we show the robot programming languages that have been developed based on the framework presented above. The linker was added to the tables although it is not a robot programming language. The reason is, that the linker itself uses common parts of the system units parser and AST generation, context analysis and code generation.

As can be seen, the framework handles robot programming languages with different characteristics equally well: PASCAL-like languages (KRL, RAPID), BASIC-like languages (MBA4), NC-like languages (ROSIV) and proprietary style languages (V+).

Table 2 and 3 show the gain of time in development and maintenance for these robot programming languages. The "estimated time" reflects the estimated time to develop, respectively to maintain a translator for this robot programming language without the framework, based on our experiences from table 1. The "development time", respectively the "actual time" reflects the real time, that was needed to develop/maintain the translator. The development times of the KRL and RAPID translators contain the time to develop the framework, otherwise they would be much shorter. The overall reduction in development time is about 70% to 80% on average. The gain in maintenance time is also about 80%. In total, the gain is higher than the proportion of use of common functionality (see table 4). The reason is, that the framework offers templates to fill in, rather than forcing the developer to program things anew.

Language (manufacturer)	*Estimated time*	*Develop. time*	*Relative gain*
KRL (KUKA)	100%	75%[1)]	↓ - 25 %
RAPID (ABB)	100%	83%[1)]	↓ - 17 %
V+ (Adept)	100%	29%	↓ - 71 %
ROSIV (Reis)	90%	25%	↓ - 72 %
MBA4 (Mitsub.)	100%	17%	↓ - 83 %
Linker	*25%*	*3%*	↓ *- 88 %*

Table 2: Compiler development time (in % of IRL)

Table 4 shows the share of common and specific parts in each system unit for the translators. The management and helper functionality (see figure 2) is not listed, because all translators use almost 100% of it. Due to this omission, the total share of common and specific parts cannot be calculated from the unit entries. Evidently, the share of common functionality increases with every translation step. On average, the translators use 2/3 of common functionality and 1/3 of language specific code.

Language	*Estimated time*	*Actual time*	*Rel. gain*
KRL	100%	30%	↓ - 70 %
RAPID	100%	15%	↓ - 85 %
V+	80%	25%	↓ - 69 %
ROSIV	75%	15%	↓ - 80 %
MBA4	130%	30%	↓ - 77 %
Linker	*100%*	*35%*	↓ *- 65 %*

Table 3: Maintenance time (in % of IRL)

Another conclusion can also be drawn: developers, who have a longer framework experience, can achieve a further gain (e.g. >80% for MBA4). But even framework novices can achieve a high gain in productivity (e.g. about 70% for V+).

[1)] The framework presented in this article has been developed as a side product of the KRL and RAPID translators. So, both translators could already profit from its advantages.

	Syntax		Context		Transformation		Code Generation		Total	
	common	specific	common	specific	common	specific	common	specific	common	specific
KRL	59.4%	40.6%	56.8%	43.2%	80.6%	19.4%	93.8%	6.2%	70%	30%
RAPID	51.5%	48.5%	56.3%	43.7%	71.4%	28.6%	89.6%	10.4%	66%	34%
V+	48.1%	51.9%	49.0%	51.0%	76.1%	23.9%	92.2%	7.8%	64%	36%
ROSIV	59.8%	40.2%	65.5%	34.5%	80.7%	19.3%	94.4%	5.6%	65%	35%
MBA4	46.7%	43.3%	66.7%	33.3%	73.2%	26.8%	89.2%	10.8%	67%	33%
Linker	*49.6%*	*50.4%*	*29.8%*	*60.2%*	-	-	*30.1%*	*69.9%*	*65%*	*35%*

Table 4: Programming effort with framework (in percent of total lines of code per step)

5 Summary

This article presents a software architecture to facilitate the development of translators for robot programming languages. This framework does not share the aims of other general compiler construction tools, like e.g. [7], [8]. Such systems can be used to implement our framework. Residing on top of these construction kits, our system abstracts the development of robot language translators. This abstraction from application specific languages offers the developer e.g. fill-in templates, common functionality, design patterns and recipes to build translators. He can therefore concentrate on the robot specific programming language elements, instead of a complete programming language in general, as it has been the case up to now [9]. We have shown that in this manner, our framework achieves an enormous reduction of development time.

The framework encapsulates almost all built-in elements of a specific language in separate files. These built-in elements usually cause most of the code changes within the product life cycle. Our approach allows to perform these changes without changing binary executables. Therefore, a considerable reduction of maintenance time has been achieved.

6 Commercial Applications

The translators that were built on basis of this software architecture, have been integrated with the commercial robot simulation system COSIMIR® (Cell Oriented Simulation of Industrial Robots) [10]. This 3D-simulation system has been developed at the IRF and is in global, practical use.

COSIROP is the combined online and offline programming software for Mitsubishi robots. It includes our translators to check the syntax of robot programs before they are downloaded to the robot controller.

References

[1] R. Dillmann, M. Huck: "Informationsverarbeitung in der Robotik," p. 124, Springer-Verlag, 1991.

[2] S. Hesse: "Industrieroboterpraxis: automatisierte Handhabung in der Fertigung," p. 201 ff, Vieweg, 1998

[3] "CRx Controller: Instruction Manual," *BFP-A5992*, Mitsubishi Electric Corporation, 1999

[4] "IRDATA - Industrial Robot Data," *DIN 66314*, Beuth-Verlag, 1994

[5] "ICR - Intermediate Codes for Robots," *ISO/CD 10562-2*

[6] A. V. Aho, R. Sethi, and J. D. Ullman: "Compilers: Principles, Techniques and Tools," Addison-Wesley, 1987

[7] F. W. Schröer: "The GENTLE Compiler Construction System," Oldenbourg-Verlag, 1997

[8] E. M. Gagnon, L. J. Hendren: "SableCC, an Object-Oriented Compiler Framework," *Proc. on Technology of O-O Lang.*, p. 140-154, Aug. 1998

[9] M. Rackovic: "Construction of a Translator for Robot-Programming Languages," *Jnl. on Intell. & Robotic Systems*, p. 209-232, Feb. 1996

[10] E. Freund, J. Rossmann: "Systems Approach to Robotics and Automation," *IEEE Int. Conf. on Robotics and Automation*, p. 3-14, May 1995