Implementation of a simulation system in the PIN language

Petr Přikryl
Department of Computer Science and Engineering,
Faculty of Electrical Engineering and Computer Science,
Technical University of Brno,
Božetěchova 2, 612 66 Brno, Czech Republic
Tel. +42-(0)5-7575 214 Fax +42-(0)5-4121 1141
E-mail prikryl@dcse.fee.vutbr.cz

Abstract: *PIN is a proposal of the language for parallel programming. The language was designed for writing hardware independent parallel programs on the high level of abstraction [5], and it is based on the generative communication (GC) mechanism [1]. The articles [3] and [4] had shown how the implementation of a simple simulation system can be parallelized using the original Linda language, and how it can be done using the newly proposed extension of Linda, respectively. This article is the third-in-the-row, and it shows how the object-oriented approach of exploitation of the GC mechanism (talking about PIN) can help to avoid practical problems.*

Keywords: Event-driven simulation, parallel programming, generative communication, Linda, PIN, object-oriented approach.

1 Introduction

PIN is a language that is still in the early stage of development. The mentioned task—implementation of a classical simulation system—could be viewed even as a testbed for development of the language itself. It was also used in my previous articles [3, 4] for testing of features of the Linda[1] language and for development of the proposed extension of the language.

In the context of a previous paragraph, I should mention that the article is more about the PIN language than about simulation. Moreover, PIN is designed as a general language, not a special language for simulation. My goal is to create purely object-oriented language, as clean as possible, as intuitive as possible, and as less restrictive for usage as possible. If the work is finished successfully, then the language will be very suitable *also* for building a simulation system.

Having the fact of limited space for the article, I will show only the core of PIN — the newly proposed language. Describing implementation of parts of a simulation system would be too paper-space consuming; moreover, it would not make sense to do it without description of syntax of PIN's operations. As this is the first article about PIN that is published, I will rather focus on description of the language, and the implementation of basic blocks of a simulation system will be shown during the presentation.

[1] A registered trademark of Scientific Computing Associates, New Haven, CT. The language was the first implementation of ideas of the generative communication mechanism.

2 The language PIN

The language design is always a trade-off between a strict mathematical purity, potentials of current hardware and software technology, and subjective demands of programmers. The language should be clean, simple, expressive, and intuitive. Extremes in some direction lead to inefficiency. If this holds for sequential programming languages then it is even more important for parallel programming tools.

One of the problems that makes parallel programming difficult is related to cooperation between parallelly running processes. Processes have to communicate somehow. The best fitting strategy—the most efficient from the computer's point of view—for nowadays computers with few processors is the explicit communication between processes. One process directly states where the message is to be sent. The opposite process explicitly waits for the message from the mentioned first process. However, building of such parallel programs is extremely exhaustive for a programmer. The reason is that he/she has to keep in his/her mind the mental picture of all the program with all dependencies between the processes. And also, he/she has to follow some general rules, when and how the processes have to behave mutually. If code of each of the processes can be viewed as a sequential one, we have to use all of our experience for sequential programming and yet something more. The "outer environment" of processes complicates rules even for programming of sequential parts of the program.

The attempt to avoid this kind of programming with all mutual dependencies of processes in programers mind led to the invention of the *generative communication* paradigm (GC) — appeared at Yale University in the 80's [1]. GC separates processes by removing the need of the *explicit* communication. Therefore, a programmer can develop processes separately — the programming becomes mentally easier. The GC paradigm allows the design of a parallel programming tool built from two separable means: a *computing language* for writing the sequential parts of a code, and the *coordination language* for putting the sequential things running together in parallel.

The PIN coordination language was derived from the Linda language (see [5]). But it tries to go a step ahead. It is designed as a purely object-oriented language. The author feels that the objects (as a category) are much more general than Linda's tuples. They are also less restrictive from the position of future needs. Linda's *tuple space* was replaced by the *object space*. The object space is an object itself (it will be referenced here as `OSp`) and the original Linda operations were transformed into methods of the object space. A purely associative access to data in Linda's tuple space brought non-trivial implementation problems that lead to the proposal of new features of tuple space — the extension of Linda, with new operations for direct access to uniquely identified tuples [4]. But the solution was not clean and clear enough — it still raised too many questions, concerning the real and safe programming. The additional operations of the extension were also adopted to PIN. The author believes that the way how the transition to PIN was done makes the things cleaner again while allowing further improvements of utilizing the generative communication mechanism. The following items describe how the original operations of Linda were transformed to PIN:

`OSp.out(`*`class`*`([`*`arguments`*`]))` — this operation is derived from the original `out()`. Similarly as in the Linda's case, the operation *creates* new object. In the object-oriented approach it means that the constructor of given class is called. The operation returns the identification of the newly created object just after constructor has finished his job. If the constructor calls any blocking operations of `OSp`, then also `OSp.out()` will block (compare it with `OSp.eval()`).

`OSp.rd(`*`class`*`[.`*`m-method`*`([`*`arguments`*`])])` — the descendant of the original operation `rd()`. The associative matching of a tuple-template with a tuple in tuple space is replaced by searching for an object of given class (it resembles usage of arity and type signature of the tuple being searched for), and by calling *match method* (m-method) with given arguments (it replaces the associative key formed by values of some of the elements of the tuple being searched for). The m-method is any method of the class that returns a value that can be interpreted as the *true*/*false* value. The m-method is called internally by `OSp` to check whether the candidate can be chosen as the object that is being searched for (*true*). Values extracted from the object are returned in arguments passed by reference — this is the reason why the `OSp.rd()` was used.

If the object is not found in `OSp`, the operation blocks until a suitable object will appear. The operation returns the identification of the object.

`OSp.in(`*`class`*`[.`*`m-method`*`([`*`arguments`*`])])` — this operation replaces the original `in()`. Similarly, as in the comparison of `rd()` and `in()` in the Linda case, the object is removed from `OSp` and even though the identification is returned, it can never be used again to identify the object (the usage would raise an error). As the operation is related to destruction of the object, its destructor is called during the act. Up to that fact, the arguments and the semantics is the same as those of `OSp.rd()`. The call in the form without a m-method can be used for removing *some* chosen object of the *class*.

`OSp.eval(`*`class`*`([`*`arguments`*`]))` — the difference between this operation and the `OSp.out()` is directly comparable with the difference between the original operations `eval()` and `out()`. The operation creates a space in `OSp` for the object being created. The constructor of the *class* is called but then the operation is finished immediately and the identification of future object is returned as the result of the operation. In other words, the `OSp.eval()` does not block in the case when the constructor does. Process that called the `OSp.eval()` and the constructor of the object being created run in parallel. It also means that `OSp.eval()` is a mean for creating parallelly running processes in a PIN program.

The following operations are derived from those of proposed in [4] as the extension of original Linda. The extension introduced so called *id-operations*, i.e. operations that accessed tuples directly via their identification. PIN also introduces splitting the object space into two parts: to the associatively accessed space — *a-space*, and to the directly-only accessed space — *pin-space* (see further in the text). Previously mentioned operations worked only with a-space while the following (all id-operations) can be used for accessing objects in both mentioned parts of object space. The only requirement is that the object must exist, otherwise the operation raises an error.

`OSp.callid(`*`id`*`, `*`e-method`*`(`*`arguments`*`))` — this operation is a direct descendant of `rdid()` from extended Linda. The method is not used for matching. It is a method for *evaluation* of arguments (e-method) and returning back the information. The object is accessed via identification *id* and the e-method can modify the data inside the object. Therefore the operation was renamed from `rdid()` to more appropriate `OSp.callid()`. This way it also replaces the operation `modid()` from the proposed extension of Linda.

It seems that it does not make sense to use the form without calling e-method because the operation itself returns again the identification of the object. As the information can be returned only via arguments passed by reference, it also seems that it does not make sense to omit arguments. Probably the only exception of usage of the call in the form `OSp.callid(id)` is the case when a process that wants to be synchronized in time with the event when the object, created by `OSp.eval()`, was finally initialized. But closer observation shows that it also does not make much sense.

`OSp.inid(`*`id`*`[, `*`e-method`*`(`*`arguments`*`)])` — semantics of the operation is derived from `inid()`. The object is destroyed in the same way like in the `OSp.in()` case, but it is accessed directly via its identification. Therefore the method is not used for matching (m-method) but exclusively for evaluation of arguments (e-method).

Unlike in the `OSp.callid()` case, it makes sense to use the form `OSp.inid(id)` (without e-method). It is the most effective way how to destroy the object.

The methods `OSp.rd()` and `OSp.callid()` were combined to provide, as author believes, another useful operation `OSp.call()`. It works over an a-space of `OSp`.

`OSp.call(`*`class`*`[.`*`m-method`*`([`*`m-arguments`*`])], `*`e-method`*`(`*`e-arguments`*`))` — the operation combines associative searching for an object (it uses m-method like in `OSp.rd()`) with access to its internals via e-method. It does not make sense to omit e-method as we would obtain exactly the functionality of the `OSp.rd()` operation. As the author does not like duplicity of that kind inside the language, the form without e-method is forbidden. If the m-method is omitted, then the one of the objects of the given class is chosen and its e-method is called.

Why the operation was ever introduced? The reason is called *efficiency* in real implementation. The m-method is expected to be very simple and quick to allow `OSp` quick testing of candidate objects. It is recommended that the m-method uses only internal data of the object—no references to other object—thus making the operation quick. The m-arguments are expected to be passed by value or, better to say, they are only input arguments. Therefore, the operation of returning back the output m-arguments need not to be performed.

The e-method is expected to be much complex and therefore slower in real implementation. But, in comparison with m-method of `OSp.rd()`, the e-method is called only once, for the object that was already chosen.

To summarize: the `OSp.rd()` is useful in cases when m-method is simple and quick enough for testing the possible candidates. It can possibly return some value, but it is preferred if the value can be extracted from the internal data of the chosen object. It can be, e.g., useful for obtaining an identification of the associatively accessed object. The operation `OSp.callid()` can be used only in the case when we know the identification of the object. But then its e-method can be very complex as it is called only once and it can be evaluated in parallel, independently on other activities inside `OSp`. The `OSp.call()` operation should be used whenever the m-method of `OSp.rd()` would be too complex, hence the operation would be too slow. Using the `OSp.call()`, the former complex m-method can be split to a simple, hence quick m-method, and to fairly complex e-method that is called only once.

Linda language with its tuple space, tuples, and operations brings to the real applications a problem that can be called an *associative access confusion* error. Many Linda applications tend to use very simple, hence very similar tuples like `(string, integer)`. Then, for example, in the case when two instances of the same application are launched in the same system and when associative links are used to build a graph data structures, the situation leads to the mixing of should-be-independent graph data structures. This can be called "*more than enough*" confusion error (talking about quantity of candidate tuples).

The id-operations introduced by [4] can help in the case when the application uses exclusively the access to tuples via identifications, e.g. for building graph data structures. But it is too restrictive. Moreover, it still gives another application the possibility to remove a tuple by an associative operation. If the removed tuple was a part of a graph data structure with explicit linkage, then the graph structure is irreversibly damaged. This can be called "*less than enough*" (of tuples) confusion error.

To avoid the confusion errors mentioned earlier, the program should be designed so that it anticipates such cases. This makes parallel programming more difficult again. Programmer can do mistakes easily because he/she has to think about distinguishing tuples created by different instances of the same application. Therefore, some possibilities for restriction of the access to only some data should exist.

PIN introduces new feature that had given the language the name and that allows a protection against the confusion error. Each object can be *pinned* down (like a butterfly). Pinned objects are ignored by the associative operations. In other words, their m-methods could always return $false$ if the object is pinned. The pinned objects can be accessed only directly, via their identification.

We can look at the situation from another point of view — the object space is separated to two subspaces: a-space (similar to Linda's tuple space) and pin-space. If any graph structure is created in the pin-space, it must be a graph structure with *explicit* links. Because of this, not only the "more than enough" confusion cannot happen, but also "less that enough" confusion error is impossible as no associative operation can unintentionally remove any object from pin-space. It can be done intentionally, of course, but the intention of a "agressor" cannot be distinguished from the intention of a "good" entity. Is should be, however, mentioned here that it is not so easy for agressor to pretend that it is a good entity. As mentioned earlier, every associative operation uses class of the object and m-methods for associative searching. Even if the agressor would be able to get the identification of the class, the PIN language can be improved so that, besides m-methods, also checking the registered identifications of applications (which are allowed to access the objects) can be switched on. The author believes that this can be more transparent solution of the problem than, for example, introduction of multiple tuple spaces in [2]. Let us describe new operations that work with pin-space:

`OSp.pin(`*`class`*`[.`*`m-method`*`([`*`arguments`*`])])` — the operation "pins down" the object that was found associatively. It means that it is moved to pin-space and it cannot be accessed associatively by the above mentioned operations. Meaning of the arguments of `OSp.pin()` is the same as of `OSp.rd()`, for example. The operation returns the identification of the pinned object.

`OSp.pinid(`*`id`*`)` — this operation works similarly like `OSp.pin()`, but the only candidate object is found via its identification.

`OSp.unpinid(`*`id`*`)` — the meaning of the operation is obvious. The identified object is "unpinned" which means that it is moved from pin-space to a-space. It can be located by the usual associative-access operations from now on.

`OSp.pinout()`

`OSp.pineval()` — the operations use the same arguments as operations `OSp.out()` or `OSp.eval()` respectively. They have also the same semantics, but they are created inside the pin-space instead of a-space. It is similar to usage of sequence of `OSp.out()` plus `OSp.pinid()` or `OSp.eval()` plus `OSp.pinid()` respectively, but the operations are atomic, i.e. nobody can steal the new object before it is pinned down.

`OSp.pinrd(`*`class`*`[.`*`m-method`*`([`*`arguments`*`])])` — this is the only operation that can be used for associative searching inside an only inside the pin-space. Up to this difference, it works exactly like `OSp.rd()` in a-space. The operation can be useful for example in the case when we want to access an entry point of a graph data structure (e.g. the root of a binary tree).

That is almost all of important features of the PIN language. However, one more mechanism introduced by PIN should be at least mentioned — the locking mechanism. Another set of new operations is designed for locking the exclusive access to objects. A locked object (almost) disappear from both a-space and pin-space. To simplify a little, only the one who locked the object can unlock it and make it visible for others again. The locking mechanism can be used for combination of `OSp` operations in an atomic way. The locking mechanism is still under development.

3 Implementation of a simulation system

In comparison with attempts in [3, 4], implementation of various parts of a simulation system is very similar to implementation in any object oriented programming language. The object oriented approach helps to hide (encapsulate) internal details of the objects thus allowing rapid development of a prototype and later improvements in implementation of classes.

4 Conclusion

Features of the newly designed, object oriented coordination language for parallel programming were described by the semantics of new operations. The language is expected to overcome weak points of classical Linda-like systems while still preserving their nice features. The author believes that the implementation of the language will not be extremely difficult, and that the system will be efficient in a real computing environment.

The basic features of the language and some other details will be presented on the example of a possible parallel implementation of a general, event-driven simulation system.

References

[1] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[2] David Gelernter. Multiple tuple spaces in Linda. In G. Goos and J. Hartmanis, editors, *PARLE'89: Parallel Architectures and Languages Europe, Vol. II: Parallel Languages*, number 366 in Lecture Notes in Computer Science, pages 20–27. Berlin: Springer-Verlag, June 1989.

[3] Petr Přikryl. Discrete simulation in the Linda programming language. In Jan Štefan, editor, *Proceedings of 5th Moravo-Silesian International Symposium on Modelling and Simulation of Systems*, pages 238–245, Olomouc, Czech republic, June 1–4 1993.

[4] Petr Přikryl. Extension of Linda and the discrete simulation. In Jan Štefan, editor, *Proceedings of 6th Moravo-Silesian International Symposium on Modelling and Simulation of Systems (MOSIS '94)*, pages 188–195, Zábřeh na Moravě, Czech republic, May 31 – June 2 1994.

[5] Petr Přikryl. PIN — the language for parallel/distributed environment. In Jan Štefan, editor, *Proceedings of 30th Spring International Conference on Modelling and Systems Simulation (MOSIS '96)*, volume 1, pages 187–192, Krnov, Czech Republic, April 23–25 1996. MARQ — Ostrava.