Delayed and Periodic Execution of Tasks in the Jadescript Programming Language

Giuseppe Petrosino[1], Stefania Monica[1], and Federico Bergenti[2]

[1] Dipartimento di Scienze e Metodi dell'Ingegneria
Università degli Studi di Modena e Reggio Emilia
42122 Reggio Emilia, Italy
{giuseppe.petrosino,stefania.monica}@unimore.it
[2] Dipartimento di Scienze Matematiche, Fisiche e Informatiche
Università degli Studi di Parma
43124 Parma, Italy
federico.bergenti@unipr.it

Abstract. Software agents are expected to timely act and to dynamically plan their activities, for example, to solve the complex collaboration problems of many real-world applications. The collaboration among agents requires the ability to reason about time to dynamically coordinate and to effectively adjust the frequency of periodic actions and reactions. For these and related reasons, an agent-oriented programming language is demanded to provide the programmer with effective means to schedule the execution of delayed and periodic tasks. This paper describes the new datatypes, and the related changes to some language constructs, that have been recently added to the Jadescript programming language to allow agents to effectively manage the dynamic scheduling of delayed and periodic tasks.

Keywords: Agent-based software systems · Agent-oriented programming · Jadescript · JADE

1 Introduction

The ability to reason about time and to act in a timely manner is of primary importance for software agents because agents are normally expected to dynamically plan and to timely act in complex and dynamic worlds, especially when real-world applications are concerned. The agents that are immersed in a *Multi-Agent System* (*MAS*) are expected to reason about time, for example, to organize their activities, to perform coordinated tasks, to detect failures, to achieve dynamic collaborations, and to react to unexpected or erroneous conducts of peers. Actually, the ability to reason about time is essential in several agent-based applications, for example, in robotics, in cyber-physical systems, and in all applications that rely on human-machine interactions. For these and related reasons, an *Agent-Oriented Programming Language* (*AOPL*) (e.g., [21]) is demanded to natively provide abstractions to support the effective development of software agents that can act in a timely manner.

Jadescript (e.g., [7]) is an AOPL that has been recently designed on top of JADE (e.g., [3]), the Java framework for the development of MASs that strongly contributed to shape *Agent-Oriented Software Engineering* (*AOSE*) (e.g., [5]). Some of the most relevant abstractions that JADE contributed to AOSE, as briefly discussed in [4], were reworked in Jadescript to have them natively embedded in the language to ensure their easy and profitable use in the daily routine of the programmer. Moreover, the availability of these abstractions in terms of native language constructs guarantees that the Jadescript compiler can block erroneous uses and possibly suggest corrections and improvements. The programmer that uses Jadescript is encouraged to think about agents and MASs in terms of agent-oriented abstractions to effectively profit from the relevant characteristics of agents in terms of reusability and composability (e.g. [6]).

However, these abstractions, that JADE provides by means of Java classes, interfaces, objects, and methods, needed to be redesigned to become an integral part of a new language and of its peculiar approach to *Agent-Oriented Programming* (*AOP*) (e.g., [21]). For example, the programmer that uses Jadescript is supposed to manage application-specific data using ontologies [17], to implement the interactions among agents using the supported speech acts [16], and to program tasks using behaviours [14]. For these reasons, the most appropriate approach to introduce delayed and periodic tasks in Jadescript was to extend the language and to enrich its support for behaviours with the possibility of dynamically selecting when a behaviour should be scheduled. This approach to the support for delayed and periodic tasks in Jadescript also required to extend the language with an appropriate support for time-related datatypes. Jadescript agents are now capable of treating time-related abstractions using dedicated datatypes that are deeply embedded in the core of the language.

The major contributions of this paper are:

1. To briefly describe Jadescript behaviours and how they are scheduled (see Section 2);
2. To present the new datatypes that were added to the language to natively manipulate time-related abstractions (see Section 3);
3. To discuss the changes introduced in Jadescript and in its underlying behaviour scheduling mechanism to support delayed and periodic tasks (see Section 4);
4. To concisely describe how other AOPLs support time-related abstractions and the manipulation of delayed and periodic tasks (see Section 5); and
5. To motivate and shortly overview possible developments of the discussed work (see Section 6).

Note that, even if the novel support for delayed and periodic tasks is a relevant improvement of the language, Jadescript and JADE are not designed for realtime applications, and these new features of the language should not be considered as an oversimplified support for realtime applications. Actually, JADE and, therefore, Jadescript are not yet ready for realtime applications, even if the literature documents interesting proposals (e.g. [13]).

2 Jadescript Behaviours

Jadescript provides behaviours to program the tasks of agents. Agents perform several tasks during their lifecycles, and therefore they activate and deactivate several behaviours. When a behaviour is activated, a reference to the behaviour is kept in a list, internal to the agent, of all activated and *ready* behaviours. An agent, during its execution loop, performs one execution of each one of its ready behaviours following a round robin scheduling algorithm. The scheduling of behaviours is said to be cooperative because behaviours explicitly release the control back to the scheduler at the end of each execution. This approach to the scheduling of behaviours has its roots in JADE, and, despite the lack of parallelism in the execution of behaviours within an agent, it has several advantages that are well known to the programmers that use JADE. The most important advantage of this approach is that the programmer is not required to take into consideration the synchronized access to the state of the agent, which is shared among its *active* behaviours. Moreover, this approach is intended to promote parallelism by distributing tasks among agents, which can cooperate and coordinate using message passing.

Jadescript supports two types of behaviours. *One-shot* behaviours are executed only once, and then they are automatically deactivated. On the other hand, *cyclic* behaviours are kept in the internal list of the active behaviours of the agent after each execution. Therefore, agents reschedule cyclic behaviours indefinitely, until explicit deactivation.

Behaviours can be created and initialized dynamically, and if a behaviour requires initialization arguments, the Jadescript compiler forces the programmer to provide the needed arguments. A created and initialized behavior can be activated by means of the `activate` statement. The related `deactivate` statement is used to deactivate an active behaviour. Behaviours can be deactivated and reactivated indefinitely, and between a deactivation and a subsequent activation, behaviours keep their internal states. Finally, Jadescript provides the `destroy` statement to deactivate a behaviour and invalidate all its subsequent activations.

Behaviour declarations are used to define new types of behaviours. Behaviour declarations can include, among other features like properties, functions, and procedures, several event handlers to provide the procedural code to be executed to react to events. Immediately after the creation of a behaviour, the `on create` event handler of the newly created behaviour is executed. This event handler can declare a list of typed parameters that can be used to initialize the state of the behaviour. When a behaviour is activated, the `on activate` event handler is executed. Note that the `on activate` event handler is executed exactly once for each activation of the behaviour. The `on execute` event handler can be used to provide the procedural code to be executed when a behaviour is scheduled. When a one-shot behaviour is deactivated after its single execution, its `on deactivate` event handler is executed. This also happens when a cyclic behaviour is explicitly deactivated by means of the `deactivate` statement. The `on destroy` event handler is executed when a behaviour is explicitly destroyed. Finally, the `on message` event handler is particularly relevant because it is of pri-

mary importance to support message passing among agents. It is executed when an agent successfully extracts a message from its inbox, thus allowing agents to use behaviours to handle the reception of messages. An `on message` event handler can restrict the set of accepted messages by explicitly mentioning the supported performative and by providing details on accepted message contents using pattern matching [15]. A behaviour can declare several `on message` event handlers, and, when scheduled, the behaviour checks if a message in the inbox of the agent matches the requirements of the available handlers. Only the first handler that matches a message in the inbox is used, and only one message is processed when the behaviour is scheduled. Therefore, a behaviour needs to be scheduled several times to process several messages in the inbox of the agent. The order in which event handlers appear in a behaviour declaration fixes, from top to bottom, the priority of the declared handlers.

In summary, the execution of the behaviours of a Jadescript agent can be outlined as follows:

1. The `on activate` event handler is executed, if available and if this is the first time that the behaviour is scheduled after its activation;
2. The `on execute` event handler is executed, if available; and
3. The first `on message` event handler that matches a message in the inbox of the agent is executed, using the priority that is implicitly assigned by the behaviour declaration.

If, in a cyclic behaviour, no event handlers can be executed, the behaviour is automatically marked as *waiting*. Waiting behaviours are still considered as active by the agent, and they are rescheduled as soon as an interesting event occurs. As a matter of fact, when a message is added to the inbox of the agent, the agent sets all its waiting behaviours as ready to allow them to check if one of their `on message` event handlers matches the new message. This mechanism is completely transparent to the programmer that uses Jadescript.

Fig. 1 shows an example of a behaviour declaration. The behaviour is intended to handle the reception of proposals in a contract-net protocol (e.g., [18]). Actually, the behaviour handles proposals by activating an `Evaluate` behaviour with the needed argument `proposal` set to the content of the message. The behaviour handles refusals by simply tracing the event in the message log of the agent. Similarly, the behaviour handles all other messages by tracing them as not understood in the message log of the agent.

3 Time-Related Datatypes

Software agents are expected to timely act and to dynamically plan their activities in real-world applications, and therefore a good AOPL is demanded to provide effective means to manipulate time-related abstractions. Jadescript provides a set of language constructs that allows the programmer to adequately manipulate time-related abstractions. Most importantly, Jadescript includes two datatypes for the manipulation of time [17].

```
cyclic behaviour HandleProposals
  for agent ContractNetInitiator

  on message propose do
    activate Evaluate(proposal=content of message)

  on message refuse do
    log sender of message + " refused."

  on message do
    log message + " received but not understood."
```

Fig. 1. Example of a behaviour declaration written in Jadescript to manage proposals in a contract-net protocol (handlers for propose and refuse messages) and to trace not understood messages (last handler).

The first datatype, called `timestamp`, is used to represent instants with a precision of one millisecond. Timestamps can be created using builtin functions like `today` and `now`. In addition, they can be created using timestamp literals, which are composed of year, month, day, hour, minute, second, and millisecond values together with a timezone offset, and they are arranged in a format that is strongly inspired by the ISO 8601 standard. Note that the values returned by `now` and `today` are related to the clock of the agent container that is currently hosting the agent. Therefore, all agents that execute in the same agent container have their internal clocks synchronized because they share the same clock. On the contrary, agents that execute in different agent containers need to adopt application-specific methods to synchronize their clocks, whenever needed. The `timestamp` datatype provides a set of builtin operations, like relational and equality operators. Two timestamps can be subtracted to have a duration (see below), and a duration can be added to or subtracted from a timestamp to have a new timestamp shifted forward or backward by the specified offset.

The second datatype, called `duration`, is used to manage time intervals. Durations support a precision of one millisecond, and they can be created using duration literals or with operations on timestamps. Duration literals are composed of day, hour, minute, second, and millisecond values followed by their respective prefixes, which are `days` (or `d`), `hours` (or `h`), `min` (or `minutes`, or `m`), `secs` (or `seconds`, or `s`), and `ms`. Any part of a duration literal can be omitted when its value is zero. Durations can be multiplied and divided by integer and real factors. Any duration can be added to and subtracted from any other duration. Moreover, a real number can be computed as the ratio of two durations. Finally, just like timestamps, durations can be compared using the provided relational and equality operators.

The parts that constitute a duration or a timestamp can be accessed as readonly properties using the `of` operator [7]. Finally, durations and timestamps can be used in pattern matching [15].

4 Delayed and Periodic Behaviours

Jadescript behaviours and their scheduling mechanism have been recently improved to support the execution of delayed and periodic behaviours. In particular, the `activate` and `deactivate` statements were extended to support a set of optional parts specifically designed to manage delayed and periodic activations, and delayed deactivations, of behaviours.

The optional parts of the `activate` statement that start with `at` and `after` allow the programmer to express the delayed activation of a behaviour. Specifically, `at` is followed by an expression denoting a timestamp, and it indicates the instant at which the behaviour will be first activated, and therefore, at which its `on activate` event handler will be scheduled. Note that if the timestamp indicates an instant before the execution of the `activate` statement, the behaviour is executed as soon as possible. Similarly, `after` is followed by a duration that indicates the delay between the execution of the `activate` statement and the first execution of the behaviour. These two options to specify when the first activation of a behaviour will occur are strongly related, and they are normally said to declare a delayed activation. Actually, they are mutually exclusive and the compiler treats `activate` B `at` T as `activate` B `after` T `- now`.

The `activate` statement can also include `every`, which is available only for cyclic behaviours. This optional part of the statement is followed by an expression that denotes a duration. The statement activates the behaviour in periodic mode to allow the programmer to declare periodic tasks. When in periodic mode, the specified duration is used as the minimum time interval that must occur between two successive executions of the behaviour.

Note that delayed and periodic activations impose constraints on the release time of a task, where the release time of a task can be broadly defined as the time at which the task becomes available for execution. Actually, delayed and periodic activations do not set a deadline on the execution of the task. Therefore, a delayed or periodic behaviour is guaranteed not to be scheduled before the specified time, but it can be actually scheduled after.

The `at` and `after` optional parts of the `activate` statement can be also used at the end of the `deactivate` statement. Delayed deactivations are used to ensure that a behaviour will not be scheduled after the specified instant. An immediate and appropriate use of delayed deactivations is for the creation of a deadline for waiting for a message, which is typically in response to a previous poll like, for example, in the progress of the contract-net protocol.

Delayed and periodic activations are implemented by modifying how waiting behaviours are treated by the behaviour scheduler of Jadescript agents. Actually, a waiting behaviour does not simply wait for events, but it also waits for a specified amount of time to pass. Similarly, when a behaviour is activated with `at` or `after`, the behaviour is set as waiting instead of ready. Finally, at the end of each execution of a periodic behaviour, the behaviour is set as waiting to wait for the successive periodic activation.

Similarly, delayed deactivations are implemented by adding a timestamp property to all behaviours. This property is hidden to the programmer, and

it is set by the `deactivate` statement. When the scheduler is going to select a behaviour, it first checks that the current time is not past the deactivation time. If it is, the behaviour is not scheduled, it is removed from the list of active behaviours, and its `on deactivate` event handler is executed.

The choice of letting the programmer define a delay or a period for a behaviour at the activation site rather than in the declaration of the behaviour was adopted to promote reusability and to ensure that delays and periods can be set dynamically. Reusability could have been damped by binding a delay property or a period property to a behaviour declaration. Conversely, letting the programmer define these two properties at the activation site allows the programmer to activate multiple related behaviours with different periods and/or different delays. Also, this design choice allows a behaviour to be reactivated with different periods or different delays, which can be computed at runtime to dynamically change the reactivity of agents to interesting events.

To exemplify the relevance of the mentioned additions to the language, consider the example shown in Fig. 1. According to FIPA specifications [18], the contract-net protocol assumes that the agent that takes the role of initiator is expected to set the deadline by which all participants have to reply with either a propose message or a refuse message. After that, the initiator can proceed to accept the best proposal and to reject all other proposals. To this purpose, an `on activate` event handler can be added to `HandleProposals` behaviours. This handler executes a delayed activation of a `HandleDeadlineExpired` behaviour to have it activated after the deadline has expired. When activated, the `HandleDeadlineExpired` behaviour selects the best proposal, and it appropriately replies to all agents that provided a proposal in time. Moreover, the `HandleDeadlineExpired` behaviour can treat incoming proposals as invalid, because they arrived late, and it can reply to such late proposals accordingly.

5 Related Work

SARL (e.g., [10, 20]) is a general-purpose AOPL whose runtime is based on the Janus multi-agent platform for Java. The language is advocated as agent-oriented, but it maintains strong connections to *Object-Oriented Programming* (*OOP*). As a matter of fact, SARL supports OOP features like objects, classes, and interfaces [12], and its statement and expression languages are created as extensions of the Xtend language [8], which is a dialect of Java. In SARL, tasks and their executions are modeled using two distinct facilities, implemented, respectively, by the `Behaviours` and the `Schedules` *capacities* [20]. SARL behaviours are similar to Jadescript behaviours in their external structure because the entry points of SARL behaviours are event handlers. However, the types of supported events are limited to external events and to the creation and the destruction of behaviours (`on Initialize` and `on Destroy`). Moreover, SARL behaviours lack an event handler to define a generic task performed by the behaviour (corresponding to `on activate` and `on execute` event handlers in Jadescript). All

in all, SARL behaviours are nothing but structures for the organization of the reactions of an agent to external events.

SARL provides the builtin `Schedules` capacity to schedule concurrent and timed tasks. This capacity enriches the scope of the agent body with extension methods to execute single-run tasks, to schedule delayed tasks, to launch tasks at specific instants, and to set the execution of periodic task. All these extension methods take as argument a method reference, or an Xtend lambda expression, to procedurally define the actual task. The distinction between behaviours and scheduled tasks in SARL creates a conceptual distance from the understanding of behaviours advocated by Jadescript, where a behaviour corresponds to a task that is performed by handling events and/or by actively performing actions. Finally, SARL does not provide builtin types to treat time. However, SARL agents can directly access the Java classes in the `java.time` package via Xtend.

Jason (e.g., [9]) is an implementation of AgentSpeak (e.g., [19]) created to develop cognitive agents based on the *Belief-Desire-Intention* (*BDI*) model. In Jason, the tasks performed by an agent are represented by the goals in the intention stack of the agent. Jason provides two functions to schedule tasks:

1. `at`, which is used to emit an event at the specified instant; and
2. `wait`, which is used to suspend the intention on the top of the stack for a specified number of milliseconds or until a specified event occurs.

The needed number of milliseconds is used to specify a time interval to `wait`, while a text with a specific syntax is used to specify an instant to `at`. Also, Jason supports two builtin actions to get the current date and the current time. Both actions unify the parts of the current date, or time, with integer variables.

6 Conclusion

Software agents are expected to timely act and to dynamically plan their activities, for example, to effectively coordinate and to adequately adjust the frequency of periodic actions and reactions. Therefore, a good AOPL is demanded to provide the programmer with effective means to schedule the execution of delayed and periodic tasks. This paper described the datatypes, and the related changes to some language constructs, that have been recently added to Jadescript to allow the programmer to effectively manage the dynamic scheduling of delayed and periodic behaviours.

Planned future works include an analysis of the described additions to the language from the performance point of view to compare the adopted solutions to alternative solutions documented in the literature. Moreover, the described enhancements to Jadescript are open to further improvements. For example, as hinted in Section 4, the behaviour scheduling mechanism that Jadescript currently adopts uses a strategy that does not guarantee that the average value of the frequency of a periodic behaviour matches the inverse of the requested period. Actually, the current behaviour scheduling mechanism treats the period

as the distance between the end of an execution and the beginning of the successive execution. So, the period does not take into account the time needed to perform the actual computation of the behaviour and all the delays caused by other active behaviours. These delays cannot be considered as negligible in real-world applications, and therefore the behaviour scheduling mechanism could be improved to ensure that the release time of a periodic task would be computed in a way that is not influenced by the actual execution time of behaviours.

As a further development, the opportunities offered by delayed and periodic behaviours could be explored in combination with the other features of the language related to the perception of the environment [11], for example, to program autonomous robots. Similarly, the ability of Jadescript agents to take time into account in their operations could be further extended to support real-world applications characterized by soft, or even hard, realtime constraints (e.g., [1, 2]). This development requires to retarget Jadescript to runtime platforms that can ensure compliance with real-time constraints, which is a very strong requirement for a language that is tightly coupled with Java and JADE.

Acknowledgements. This work was partially supported by the Italian Ministry of University and Research under the PRIN 2020 grant 2020TL3X8X for the project *Typeful Language Adaptation for Dynamic, Interacting and Evolving Systems* (*T-LADIES*).

References

1. Alzetta, F., Giorgini, P.: Towards a real-time BDI model for ROS 2. In: Proceedings of the 20th Workshop "From Objects to Agents" (WOA 2019). CEUR Workshop Proceedings, vol. 2404, pp. 1–7. RWTH Aachen (2019)
2. Alzetta, F., Giorgini, P., Marinoni, M., Calvaresi, D.: RT-BDI: A real-time BDI model. In: Advances in Practical Applications of Agents, Multi-Agent Systems, and Trustworthiness. The PAAMS Collection. Lecture Notes in Artificial Intelligence, vol. 12092. Springer (2020)
3. Bellifemine, F., Bergenti, F., Caire, G., Poggi, A.: JADE–A Java Agent DEvelopment Framework. In: Multi-Agent Programming, Multiagent Systems, Artificial Societies, and Simulated Organizations, vol. 25, pp. 125–147. Springer (2005)
4. Bergenti, F., Caire, G., Monica, S., Poggi, A.: The first twenty years of agent-based software development with JADE. Autonomous Agents and Multi-Agent Systems **34**(36) (2020)
5. Bergenti, F., Gleizes, M.P., Zambonelli, F. (eds.): Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook. Springer (2004)
6. Bergenti, F., Huhns, M.N.: On the use of agents as components of software systems. In: Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook. pp. 19–31. Springer (2004)
7. Bergenti, F., Monica, S., Petrosino, G.: A scripting language for practical agent-oriented programming. In: Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control

(AGERE 2018) at ACM SIGPLAN Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH 2018). pp. 62–71. ACM (2018)
8. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing (2013)
9. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming multi-agent systems in AgentSpeak using Jason. Wiley Series in Agent Technology, Wiley (2007)
10. Feraud, M., Galland, S.: First comparison of SARL to other agent-programming languages and frameworks. In: Proceedings of the 8th International Conference on Ambient Systems, Networks and Technologies (ANT 2017) and of the 7th International Conference on Sustainable Energy Information Technology (SEIT 2017). Procedia Computer Science, vol. 109. Elsevier (2017)
11. Iotti, E., Petrosino, G., Monica, S., Bergenti, F.: Exploratory experiments on programming autonomous robots in Jadescript. In: Proceedings of the 1st Workshop on Agents and Robots for Reliable Engineered Autonomy (AREA 2020) at the European Conference on Artificial Intelligence (ECAI 2020). Electronic Proceedings in Theoretical Computer Science, vol. 319. University of New South Wales (2020)
12. Najjar, A., Rodriguez, S., Zhao, H., Tchappi, I.H., Galland, S., Mualla, Y., Gaud, N.: Model transformations from the SARL agent-oriented programming language to an object-oriented programming language. International Journal of Agent-Oriented Software Engineering **7**(1) (2019)
13. Pereira Filgueiras, T., Lung, L.C., de Oliveira Rech, L.: Providing real-time scheduling for mobile agents in the JADE platform. In: Proceedings of the 15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC 2012). pp. 8–15. IEEE (2012)
14. Petrosino, G., Bergenti, F.: An introduction to the major features of a scripting language for JADE agents. In: Proceedings of the 17th Conference of the Italian Association for Artificial Intelligence (AI*IA 2018). Lecture Notes in Artificial Intelligence, vol. 11298, pp. 3–14. Springer (2018)
15. Petrosino, G., Bergenti, F.: Extending message handlers with pattern matching in the Jadescript programming language. In: Proceedings of the 20th Workshop "From Objects to Agents" (WOA 2019). CEUR Workshop Proceedings, vol. 2404, pp. 113–118. RWTH Aachen (2019)
16. Petrosino, G., Iotti, E., Monica, S., Bergenti, F.: Prototypes of productivity tools for the Jadescript programming language. In: Proceedings of the 22nd Workshop "From Objects to Agents" (WOA 2021). CEUR Workshop Proceedings, vol. 2963, pp. 14–28. RWTH Aachen (2021)
17. Petrosino, G., Iotti, E., Monica, S., Bergenti, F.: A description of the Jadescript type system. In: Proceedings of the 3rd International Conference on Distributed Artificial Intelligence (DAI 2022). Lecture Notes in Computer Science, vol. 13170, pp. 206–220. Springer (2022)
18. Poslad, S.: Specifying protocols for multi-agent systems interaction. ACM Transactions on Autonomous and Adaptive Systems **2**(4), 15:–15:24 (2007)
19. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: MAAMAW 1996: Agents Breaking Away. pp. 42–55. Springer (1996)
20. Rodriguez, S., Gaud, N., Galland, S.: SARL: A general-purpose agent-oriented programming language. In: Proceedings of the IEEE/WIC/ACM International Joint Conferences of Web Intelligence (WI 2014) and Intelligent Agent Technologies (IAT 2014). vol. 3, pp. 103–110. IEEE (2014)
21. Shoham, Y.: Agent-oriented programming. Artificial Intelligence **60**(1), 51–92 (1993)