Chapter 8
Prometheus Research Directions

Lin Padgham, John Thangarajah, and Michael Winikoff

Abstract Prometheus is a well-established and widely used methodology. In this chapter, we briefly review the methodology and then discuss a number of active research directions. The key research directions that we discuss are: automated testing of agent systems, including test coverage; development of agent systems that are structured as *teams*, and of open agent systems that operate within Electronic Institutions; and the design and representation of agent interaction. We also briefly present the Prometheus Design Tool (PDT) and conclude with a brief look at other areas for future work.

Keywords Agent-oriented software engineering • Interaction design • Prometheus • Teams and organizations • Testing

1 Introduction

Prometheus is a well-established methodology for assisting and guiding developers in building agent-based applications. Its development started in the late 1990s as a result of collaboration between academics at RMIT University who were teaching students to develop agent programs and doing research in agent systems, and practitioners who were building and marketing agent development platforms and doing commercial work in building agent based applications. In the early 2000s, work was consolidated into the named methodology, a support tool (PDT) was developed, and in 2004 a book was published [19], which provided detailed

L. Padgham (✉) • J. Thangarajah
RMIT University, Melbourne, Australia
e-mail: lin.padgham@rmit.edu.au; john.thangarajah@rmit.edu.au

M. Winikoff
University of Otago, Dunedin, New Zealand
e-mail: michael.winikoff@otago.ac.nz

O. Shehory and A. Sturm (eds.), *Agent-Oriented Software Engineering*,
DOI 10.1007/978-3-642-54432-3_8, © Springer-Verlag Berlin Heidelberg 2014

guidelines and a running example for agent system design. In 2005, PDT was demonstrated at AAMAS and won the award for best demonstration.

Today Prometheus is used quite widely internationally, though primarily for teaching and research purposes with only relatively limited use within industry. Nevertheless, it is used by some industry practitioners, including some customers of Agent-Oriented Software, the producers of the JACK™ development platform, a widely used and comprehensive commercial agent development platform [24].

We now briefly outline the three phases that constitute the core of Prometheus, after which we mention a number of topics that are not part of the core of the methodology, but where we have done some work. The remainder of this chapter focuses on three areas where we have substantial recent or ongoing research relating to Prometheus: automated testing, representation of teams and organizations, and the design of agent interactions. We then discuss the Prometheus Design Tool, before concluding with a look to the future. A more comprehensive overview of the state-of-the-art in Agent-Oriented Software Engineering can be found in [27].

1.1 Prometheus Design Phases

The core of the Prometheus methodology consists of three design phases: System Specification, Architectural design, and Detailed design. Each of these phases includes a number of design artifacts encompassing structural design, dynamic processes and detailed descriptors. Many of the artifacts are highly structured to support automated propagation of information, both between phases and between aspects of a specific phase. The use of structured information also supports a range of consistency checks to ensure the internal coherence of the design. Figure 8.1 shows the key artifacts in each phase and how they relate to other artifacts. The three phases are discussed in a little more detail below. This discussion is intended to give a high-level summary of the methodology. Since the focus of this chapter is not to introduce the methodology in detail, we do not include a detailed example, but instead refer the reader to existing examples of Prometheus designs that have been published [6, 19].

1.1.1 System Specification

During System Specification, the designer develops an overview of how the agent system fits into the context in which it will operate, and what are the interactions it has with external *actors*, which may be humans or other systems/subsystems. This is captured in an analysis overview diagram that shows the core system scenarios or functionalities and their interactions with key external actors (via *percepts* or incoming information, and *actions* where the system provides information to or otherwise influences external entities). This, together with a goal hierarchy of top-level goals and their subgoals, provides a high-level view of the structure of the system. Scenarios, similar to use cases, provide examples of key processes and

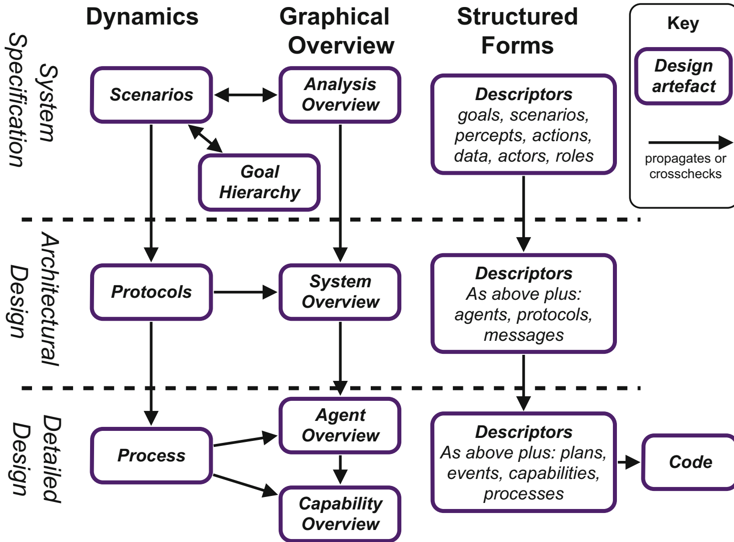

Fig. 8.1 Overview of prometheus

how they execute. These are described using a possible sequence of actions, goals, percepts, and subscenarios. Roles are also specified, and descriptors (structured forms) are developed to describe these as well as all the other entities such as percepts, actions and goals.

1.1.2 Architectural Design

During architectural design, decisions are made as to how to group the roles into agents, and which goals should belong to each agent. The communication protocols are also developed to specify the dynamics of agent interactions required to achieve the various goals. The system overview diagram captures the top-level view of the system architecture, showing which agents (agent types) exist, what communication protocols are defined for interaction between which agent types, and also any data stores within the system with shared access. Various information is also captured in descriptors for each of the entities in the design.

1.1.3 Detailed Design

During detailed design the internals of each agent type are developed to allow the agents to accomplish their specific goals or tasks within the overall system. Information from the architectural design specifies what percepts an agent is expected to respond to, what actions it is expected to take, and what messages it is expected to both receive and send as part of communication protocols. Internally

it must then do whatever processing is necessary to make choices and fulfill its role. Detailed design may contain multiple levels of hierarchy using the concept of capabilities to encapsulate modules (or collections of plans and goals), which are related to a specific aspect of the agent's functioning. An agent overview diagram captures the structure of capabilities and the interactions between them, while a capability overview diagram (at least at the lowest level) specifies which plans are designed to accomplish which subgoals, and which subgoals are part of achieving some plan. The capability overview diagram (at least at the lowest level) shows plans and their triggers (e.g., messages, internal events, or (sub-)goals),[1] where each goal has a number of plans, from which one is selected to achieve the goal based on the particular situation. Each plan may have a number of subgoals, the achievement of which is part of executing the plan. A modified version of UML activity diagrams can be used to describe the dynamic process within an agent, as part of a particular protocol, or joint effort to achieve a system goal. Descriptors are again used to provide details of each entity.

1.2 Extensions Beyond the Core

Prometheus is continually being developed and refined as a result of use, feedback, and ongoing research work. There have been a number of areas where work has been done that builds on the core. These include scoping and prioritising [18], maintenance [3], goal oriented protocols [2], testing [23, 29] debugging [20], and model driven architecture [12]. Currently, the key research areas in Prometheus are automated testing using design specifications, design of agent systems using teams and organizations, and a better representation of agent dynamics and interactions. There are also important issues around tool development and support. We will discuss each of these, outlining briefly the current research and state of the art. There are also a range of topics that are an important aspect of future work in refining and developing Prometheus and PDT, including integration with other design paradigms such as object oriented design (UML) and database design. However, many of these have to do more with industry needs than research areas, so we do not cover them in any detail here.

2 Automating Testing

Automated testing is an area where there is tremendous potential for added value within agent-oriented development environments and methodologies. Within Prometheus we have been doing substantial work in this area, with its beginnings

[1] The diagram also shows data, actions, and percepts.

about 10 years ago, but with increased focus over the last 4–5 years. The fact that Prometheus provides structured artifacts—generally produced using PDT—means that these can be used as input to software that can automate (or partially automate) testing processes. The flexible nature of agent systems means that there are very many different ways that tasks can be accomplished [26], making it very difficult to do manual testing sufficiently thoroughly. Consequently, we see automated testing as extremely important. In looking at testing there are three different essential phases to any testing process[2]: specifying the test cases, running the test cases, and checking and reporting on the correctness of the output. Our aim is to automate as much as possible of all three of these phases. There are also different levels at which a system can potentially be tested: unit testing that separately tests the component pieces, integration testing that tests how the components work together, and system testing that tests the system as a whole. We have done some work within Prometheus on each of these levels, and we have also done some initial work on how one might characterize the adequacy of a particular set of tests with respect to a specific program [16]. We describe below the work we have done at each of the testing levels and finish with a description of our work on measuring test coverage.

2.1 Unit Testing

Unit testing is the most well-developed aspect of testing within Prometheus [31], and we have a preliminary version of PDT that integrates automated unit testing, generating a comprehensive test report of issues found. This was the focus of a 2011 PhD thesis [29], and evaluation found that the approach was able to identify some intermittent and difficult-to-reproduce bugs in student written software.

In doing unit testing for an agent system, one of the issues is what are the units, or components that should be tested, as well as what aspects should/could be tested. In Prometheus we identified events (or goals), plans, and beliefs as the core component units that we test. The aspects of these components that we test are developed by considering what information is captured in a Prometheus design, and how this information can be used to test these components. For example, if a number of different plans are specified for achieving a goal, one would expect that, with a comprehensive set of tests, each plan would be used in some situation. If it is not, then, while it is not necessarily an error (it may be that we did not test the right cases, or it may be that the plan is there only as a backup in case a preferred plan fails during execution), a warning is generated to allow the developer to check. If it is the case that a suitable test was not used, the developer can add the appropriate test case, and it will be recorded for future use. If the intention is that no test case should result in the plan being chosen as the first priority, then it can be noted and

[2] An additional, nonessential, phase that we discuss below is assessing the quality of a collection of test cases, that is, coverage analysis.

the system will not notify the warning in future (unless it is requested to ignore such cases).

Another test that is related to a goal/event and a set of plans is whether, in any particular situation there is an applicable plan type. It is not necessarily wrong to have no suitable plan for achieving a goal in some situations. However, having this situation arise unexpectedly is a common cause of errors in agent programs. Developers using Prometheus are prompted to specify both *coverage* (whether there will always be some plan applicable for the event/goal) and *overlap* (whether there are any situations where there will be multiple plan types applicable—and if so how should the choice be made between them). During testing these aspects are checked, and all cases of overlap or lack of coverage are recorded for potential checking. If they are inconsistent with the specification they are flagged as an error.

Nineteen different aspects were identified that could be tested with respect to goals, plans and beliefs. The testing tool developed annotates program code automatically, to provide a test driver for each component that can automatically be executed with a suite of tests, collecting the data for analysis and reporting. Components are tested in an order appropriate to the hierarchical structure of the program, that is, components that are used by another component are tested prior to testing the using component. Cyclical structures are managed separately as a unit.

Generation of test cases is also an important aspect of testing. If these must be specified manually this will inevitably limit the extent of testing. In Prometheus, we require that the developer provide some information about variables, such as type and valid value ranges. On the basis of this information, we automatically generate test cases using equivalence classes of variable values. Where possible we generate test cases with all valid equivalence class combinations of values. However, if desired we can apply pairwise reduction to limit the number of test cases.

In our unit testing work, we assume that the design artifacts form the oracle that identifies correct behavior. It may of course be the case that code is functioning as desired but the design artifact is faulty. Our position here is that such errors are as important to identify and fix as bugs in code, in order to ensure ongoing consistency between design and implementation, as a support for maintenance and ongoing development.

2.2 Integration/Interaction Testing

Integration testing involves ensuring that no errors in behavior are introduced when components that are individually tested are used together within the system. Our approach to unit testing ensures integration testing within an agent, as our approach to testing abstract plans involves execution of all (previously tested) subsidiary plans. An important aspect of testing agent systems is the interaction between agents, and this is an area where errors often arise. In work on debugging [21], we have used the protocol specifications developed during design as an oracle regarding correct interaction behavior. The approach developed converts these to a

Petri net specification that is then executed as the program executes, identifying any mismatches between specification and actual execution. These are then identified as errors.

This work provides an oracle for determining correctness of behavior. However, work on automated generation of test cases for interaction testing, as well as their automated execution, is work in progress as part of a PhD thesis.

2.3 System Testing

In the area of system testing, we have currently focused on use of scenarios as the oracle against which behavior will be measured. In [23], we refined the previous specification of scenarios to provide additional information about percepts and actions, in particular their possible order of occurrence. This then enables us to use this specification to determine whether a particular test case conforms to expectations. We use a simulation system to initialize different configurations of the environment, and then to accept actions and generate percepts, allowing us to systematically test all scenarios. Automated test case generation can in principle follow the same approach as for unit testing—namely creating test cases where relevant variables are initialized to a value within each equivalence class, and combined systematically to create a comprehensive set of situations.

2.4 Test Coverage

One of the key issues in testing is how to know when you have tested sufficiently. The approach we have used in our unit testing is systematic generation of test cases according to equivalence classes of variable values. An alternative approach is to measure to what extent the test suite provides coverage of the code. In [16], we define different levels of coverage for interaction testing. We (loosely) base our coverage criteria on a notion of graph coverage, where the graphs are induced from messages within a protocol, or for more extensive coverage, on plans sending and receiving messages within a protocol, or plans within a decision chain regarding sending of such messages.

The basic idea in this work is that protocol specifications, together with detailed design of plan–event relationships, allows induction of a graph showing all possible paths from the start of a conversation, through plans and messages, to the conclusion of a conversation. Given this graph it is possible to define different levels of coverage, mapping to node, arc and path coverage in the graph. The most basic is message coverage, and requires only that every message is sent at least once. Plan coverage requires that each plan that sends or receives a message in the protocol is executed at least once. However, these are weak criteria, equating only to ensuring that every method in an interface is called, for standard integration testing. Arc

coverage is stronger than node coverage, and path coverage—which can include or exclude plan nodes between message receipt and message sending—is stronger again.

The paper [16] explores the nuances of these different coverage criteria and how they can be monitored using the Petri-net representation used for protocol testing as described above and in [21]. Using this kind of coverage criteria it is possible to keep generating test cases (possibly with some "intelligence") until an acceptable level of coverage has been reached.

3 Teams, Organizations, and Social Agents

The Prometheus methodology supports the development of multi-agent systems, where multiple agents may communicate (via message passing as specified in a protocol) to achieve their own goals as well as goals of the system. However, the current Prometheus design does not provide explicit support for specifying team and organizational structures that are specializations of multi-agent systems. Developing methodology support for agent teams is one of our current areas of research and we have developed an approach for designing agents for Electronic Institutions.

3.1 Teams

There are many existing team-based agent programming tools, for example, JACK Team,[3] Machinetta,[4] and GORITE.[5] These approaches incorporate team specific concepts such as team, role, joint-goal, shared-belief, shared-plan, and subteam, which are not found in the common agent design methodologies, including Prometheus.

We are currently investigating, by considering the popular team-based agent frameworks mentioned above, the common and necessary concepts for building agent teams. Whilst some of these concepts such as "team", will be new to the methodology, others such as "role" and "goal" exist but may require different semantics in a team environment. In addition to incorporating these concepts into the different stages of design, we will also develop suitable mechanisms for auto-propagation. For example, a "team-goal" will be propagated into the Agent design of the agents that belong to the team.

[3] www.aosgrp.com.

[4] teamcore.usc.edu/doc/Machinetta.

[5] www.intendico.com/gorite.

The result of our current research will be a *Team* plugin to the Eclipse-based (PDT, see Sect. 5) that, when enabled, provides the additional constructs and mechanisms necessary for developing teams, including automatic-code generation.

3.2 Electronic Institutions

Virtual organizations where heterogenous agents can join and interact with other agents to achieve their own individual objectives (e.g., agents in an auction house where buyers and sellers interact) are becoming more commonplace. Electronic Institutions are a way of implementing the interaction conventions for agents to regulate their interactions and establish commitments in such environments.

In [22], we developed an approach for designing agents for Electronic Institutions, by incorporating a *social design* phase developed using ISLANDER—a tool for building e-institutions—into the methodology. The approach is for the initial analysis and requirements to be developed in PDT using the Prometheus System Specification stage with additional concepts, such as *soft goals*, to incorporate the additional needs of an e-institution. This is then exported to the ISLANDER tool for developing the social design, which is the electronic institution component, where the interaction rules, norms and obligations of the various roles are specified. The social design is then imported into PDT, and the Prometheus approach is used to develop those parts of the system that lie outside the actual electronic institution infrastructure, in particular the agents that participate. The social design provides information such as interaction specification and ontology definitions that are incorporated into the architectural design with PDT.

One challenge with Electronic Institutions is that they should allow heterogenous agents to interact in an open environment. We explored some of these issues, proposing a layered architecture for enabling agents to join an e-institution [7].

4 Representing Interactions

A significant outstanding challenge concerns the design of agent interactions, and their representation. Almost any multi-agent system involves interactions between agents, and almost invariably (for the sorts of systems that AOSE methodologies deal with[6]) this interaction is realized using messages sent between agents. Unfortunately, current approaches to designing agent interactions have two significant weaknesses.

[6]There are also certain types of agent systems where interaction is realized not by messages, but through changes to the environment (“stigmergy”), such as depositing pheromones to indicate paths.

The first weakness is that the common approach for designing agent interactions is one that focuses on the messages, by describing explicitly the legal sequences of messages that are permitted in an interaction (i.e., an interaction protocol). Whilst using interaction protocols works fine for non-agent software, it is a poor match for agents [2]. This is because an interaction protocol is *prescriptive*: it prescribes the precise sequences of messages that can occur. Flexibility needs to be explicitly indicated, and is the exception, not the default behavior. On the other hand, agents are designed to be flexible in how they achieve their goals, and this flexibility allows them to be robust: if something goes wrong, a plan-based agent will try an alternative plan to realize the goal at hand. In other words, there is a mismatch between prescriptive interaction protocols based on legal sequences of messages, and flexible and robust agents. One consequence of this mismatch is that interactions designed using the common message-sequence-based approach tend to be brittle, because they do not exploit the flexibility of the agents participating in the protocol.

The solution to this first issue is to develop alternative representations for interactions that allow the flexibility of agents to be exploited, along with appropriate design processes and heuristics for using these alternative representations. A range of approaches have been proposed (e.g. [2, 8, 14, 28]), and they all have in common that they design the interaction not at the level of legal sequences of messages, but at the level of what *drives* the messages to be sent, such as goals or social commitments. By focusing on *why* an interaction is taking place, rather than on the sequence of messages, the interactions tend to be more flexible. For example, using an approach based on commitment machines [28], any sequence of messages that discharges existing commitments is permissible. However, although these approaches are promising, none of them are sufficiently well developed and refined to be currently usable for real applications. More work is needed to further develop and refine these approaches, including both concepts, and also clear and detailed methodologies, including support tools.

The second weakness relating to agent interactions and their representation is that the notations used to capture interaction protocols—such as Finite State Machines (FSMs), Petri nets, and Agent UML—are lacking features to support designers. Specifically, a good notation should provide mechanisms for *abstraction* that make it easy for the designer to decompose a design into loosely coupled aspects, and then consider each aspect in (relative) isolation. For example, in a Holonic manufacturing system [11] a rotating table needs to be locked at various points in the interaction (in order to prevent it moving while a robot is loading, unloading, or joining parts). It would be useful for a designer to be able to specify the sequence of steps involved in manufacturing a part (and associated agent interactions), separately from the details of locking. However, this cannot be specified using existing notations (such as the `ref` region in Agent UML). This is because locking involves three steps (lock, perform a task, and then unlock), and the second step is specific to the context: each time we invoke the lock sub-protocol, the second step will be a different sequence of messages. Agent UML does not have a way to pass a protocol as a parameter to a `ref` region, which means that its abstraction mechanism cannot handle this sort of situation. The lack of a good mechanism for abstraction in interactions leads to

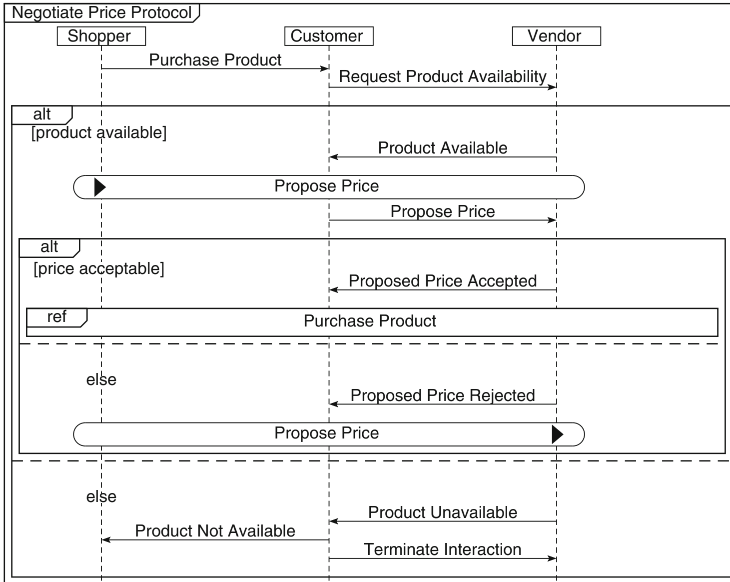

Fig. 8.2 Example interaction protocol in Agent UML

interaction protocols being difficult to specify and to understand (see Fig. 8.2 for a typical interaction protocol).

Other desirable features that are lacking in current notations include: representing agent responsibilities; showing the creation and discharge of social commitments; depicting percepts and actions[7]; clearly indicating the trigger of an interaction; and linking to the processing within each agent (e.g., the goals that it achieves at various points in the interaction [13]). These missing features also have implications for the implementation of interaction protocols. Currently, mapping a protocol into what each individual agent needs to do to play its part in the interaction, is a manual and error-prone process. Extending protocols to include information on triggers and on the agent goals that are involved would allow this process to be better supported.

Unfortunately, this second issue has not been adequately tackled in the literature. What is needed is a new notation (or significant extensions to existing notations) that allow for these various aspects to be represented; along with a revised design process. It is important for the new (or extended) notation to be precisely defined; for the design process to include detailed heuristics and clearly specified steps; and for there to be tool support, especially for the implementation step.

Taken together, these issues imply that the current state of the art in designing multi-agent interactions is adequate (at least for interactions that are not too

[7] It is fairly easy to extend AUML to do this, for instance, by depicting the environment as an additional agent.

complex), but that interaction design is not well supported by existing notations and processes, and that the interactions that are produced tend not to exploit the flexibility and robustness of individual agents.

5 Tool Development

An essential aspect of any software development methodology, is tool support. The Prometheus Design Tool is an integrated development environment (IDE) for agent system designers that offers a graphical interface for designing agent systems following the Prometheus methodology. It includes features such as type safety, entity propagation, automatic code generation, report generation and other features, some of which we describe further in this section.

PDT is developed as a plugin to the Eclipse IDE[8]—a popular open source and extensible IDE for developing software applications. Thus, PDT inherits many of the rich features of Eclipse such as file management, project management, version control, editing, coding and debugging tools that supports a variety of languages.

Figure 8.3 provides a screenshot of the PDT tool within Eclipse with the main views highlighted: *Graphical Editor* (top-center) where each diagram is displayed and edited, *Entity Palette* (top-left) for adding entities to a diagram, *Diagram Outline* (top-right) that lists the diagrams in a Prometheus design and *Entity Properties* (bottom-right) that displays the structured descriptor of a selected entity including its relationships to other entities. The tabs in this view group related attributes. For example, a data entity has a "general properties" tab (for name, description etc.), a "data fields" tab (describing the fields of the data) and an "events posted" tab that describes any events posted when the data is modified.

We highlight some of the features of the PDT tool below:

- **Graphical Editor**: All of the diagrams that are part of a Prometheus design can be graphically edited in PDT. The graphical layouts take advantage of Eclipse features such auto-arrange and a miniature view to easily navigate large diagrams. PDT also provides a drag-and-drop feature for grouping entities in an agent design into new capabilities.
- **Automatic Propagation**: As with type safety, appropriate automatic propagation of entities assists in minimising design errors and is an important support aspect of PDT. Propagation occurs in PDT when entity relations are created or modified. For example, when a role is associated to an agent, all goals achieved by the role are automatically propagated to that agent.
- **AUML Protocols**: PDT supports protocol specification via a textual protocol editor that employs a modified version of Agent UML [25]. The corresponding

[8] www.eclipse.org.

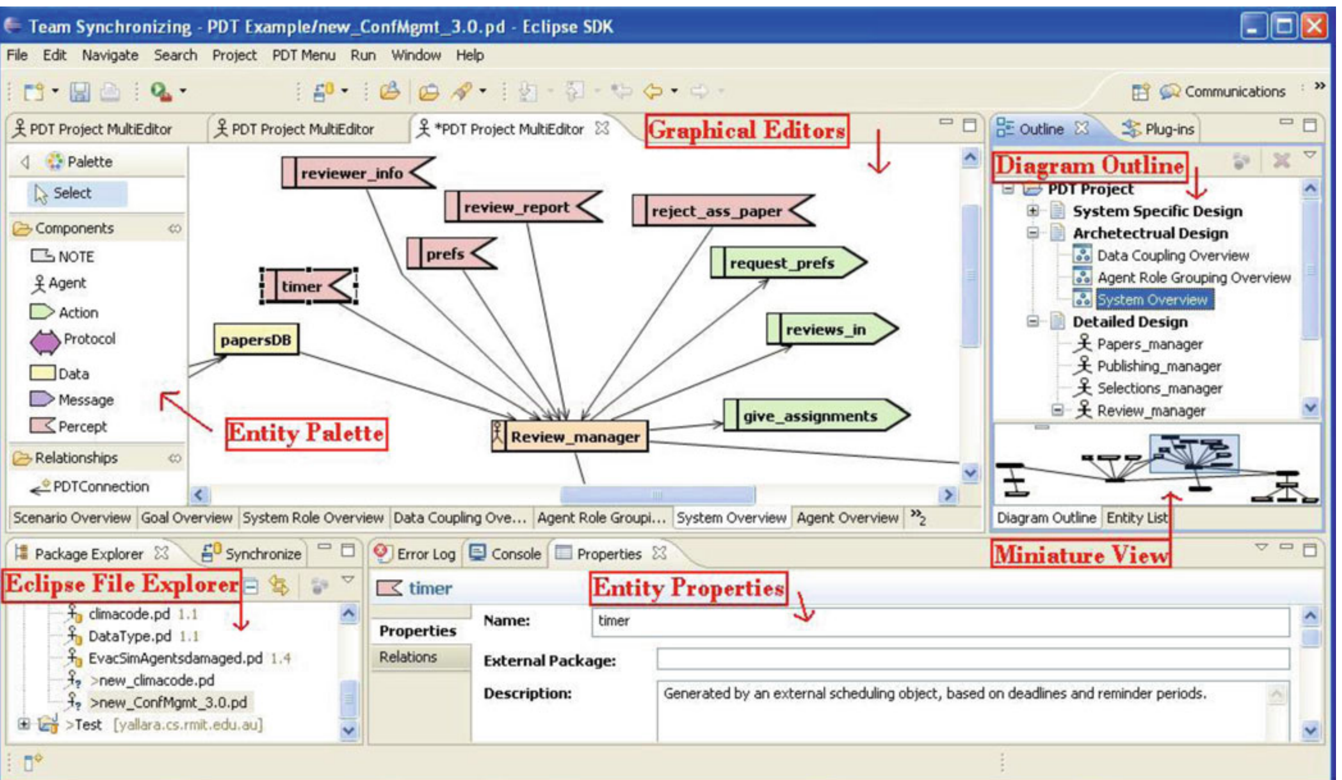

Fig. 8.3 Snapshot of PDT in eclipse

AUML interaction diagram is automatically generated by the tool. The text editor supports syntax highlighting and auto-indentation.

- **Code Generation**: This is an important aspect to assist in ensuring that the implementation is faithful to the design. In the current version, PDT is able to generate skeleton code in the JACK agent programming language,[9] using the detailed design descriptions. Developers may iterate between coding and design with manual updates to the code retained. We note that the design is such that the code generation module may be replaced to generate skeleton code into other agent programming languages. Work is currently underway to provide skeleton code generation to GORITE,[10] an open source agent programming language that we use in a number of our projects.
- **Report Generation**: PDT supports the generation of an HTML formatted report comprising all the graphical and textual information of the design, or the export of the individual diagrams, where the resolution/size of the images may be customized.

The above features are in the mainstream release of PDT that can be freely downloaded from www.cs.rmit.edu.au/pdt. The tool has attracted over 2000+ downloads since 2005. In addition to the main release, there have been other branches that have been developed by the agents group at RMIT incorporating various research aspects. Of particular significance and maturity are the PDT-UnitTesting plugin [30] and the CAFnE tool [12].

The PDT-UnitTesting plugin incorporates the techniques discussed in Sect. 2.1 and is an extension to PDT within the Eclipse framework. It adds a "test" tab to the properties view that elicits additional information necessary for testing, and menu items to perform automated testing that generate and display test reports.

The Component Agent Framework for domain-Experts (CAFnE) toolkit is an extension to an earlier version of PDT (now deprecated) that allowed complete code generation via component modules. CAFnE takes the detailed design components and first creates a domain dependent but platform independent component model of the application. This model is then transformed by a transformation module into code for a specific agent platform that can be compiled and executed.

6 Further Directions

There are a number of other potentially valuable areas of work that could use or refine/extend Prometheus, some of which are currently receiving varying degrees of attention from the AOSE community. We have covered those where we are actively

[9] www.aosgrp.com.

[10] www.intendico.com/gorite.

working, but we now briefly mention a number of additional topics, that could be fruitful areas of endeavor.

Beyond Testing: As discussed earlier, it is particularly challenging to obtain assurance that an agent system will behave appropriately in all situations. Work on formal verification (e.g., [5]) can be an alternative or complement to testing. Currently state-of-the-art model checkers for agent systems are still only applicable to toy programs, not real systems. Potentially, formal verification could be integrated into Prometheus with regard to crucial subparts of the system, or some form of partial formal verification could be done using some of the design artifacts of Prometheus.

Designing Emergent Systems: The bulk of the work on AOSE methodologies has focussed on so-called *cognitive* agents. That is, multi-agent systems where each agent has some form of reasoning capability (such as BDI agents). However, there are also agent systems that consist of very simple agents, where interesting behavior arises from the emergent interaction of many agents [1, 15]. While development environments exist for such systems (e.g. Repast [17]), there are no real methodologies for design of such. There is also potential for a design methodology for systems incorporating both cognitive and simple agents.

Software Maintenance: Once it has been implemented, software is usually subject to ongoing changes (“software evolution” or “software maintenance”) to fix bugs, add features, or deal with changes in the deployment environment. Although these changes can account for most of the cost of software, there has been very little work on software maintenance of agent software [3].

Standardization: Although the number of AOSE methodologies in active use and development has shrunk, there are still a number of methodologies. It has been argued that it would be desirable to standardize methodologies, in order to avoid gratuitous differences, and to allow further development to build on a common core [4, 10].

Industrial Adoption: Finally, the industrial adoption of agent-based solutions is being held back by various practical issues [9]. If agents are to be more widely adopted, then we need to ensure that we integrate AOSE methodologies, tools and standards with mainstream approaches. Although this work is arguably best done by industry, the academic community has a role to play.

References

1. Axelrod R (1997) The complexity of cooperation: agent-based models of competition and collaboration. Princeton University Press, New Jersey
2. Cheong C, Winikoff M (2009) Hermes: designing flexible and robust agent interactions. In: Dignum V (ed) Multi-agent systems: semantics and dynamics of organizational models, Chap. 5. IGI, Hershey, pp 105–139
3. Dam HK, Winikoff M (2011) An agent-oriented approach to change propagation in software maintenance. J Auton Agents Multi-Agent Syst 23(3):384–452. doi:10.1007/s10458-010-9163-0

4. Dam HK, Winikoff M (2013) Towards a next-generation AOSE methodology. Sci Comput Program 78:684–694 doi:10.1016/j.scico.2011.12.005
5. Dastani M, Hindriks KV, Meyer JJC (eds) (2010) Specification and verification of multi-agent systems. Springer, Berlin/Heidelberg
6. DeLoach SA, Padgham L, Perini A, Susi A, Thangarajah J (2009) Using three AOSE toolkits to develop a sample design. Int J Agent-Oriented Software Eng 3(4):416–476
7. Dignum F, Dignum V, Thangarajah J, Padgham L, Winikoff M (2008) Open agent systems. In: Agent-oriented software engineering VIII. Lecture notes in computer science, vol 4951. Springer, pp 73–87
8. Flores RA, Kremer RC (2004) A pragmatic approach to build conversation protocols using social commitments. In: Jennings NR, Sierra C, Sonenberg L, Tambe M (eds) Autonomous agents and multi-agent systems (AAMAS). ACM, pp 1242–1243
9. Georgeff M (2009) The gap between software engineering and multi-agent systems: bridging the divide. Int J Agent-Oriented Software Eng 3(4):391–396
10. Henderson-Sellers B (2010) Consolidating diagram types from several agent-oriented methodologies. In: Proceeding of the 2010 conference on new trends in Software Methodologies, Tools and Techniques (SoMeT). IOS, The Netherlands, pp 293–345
11. Jarvis J, Rönnquist R, McFarlane D, Jain L (2006) A team-based holonic approach to robotic assembly cell control. J Netw Comput Appl 29(2–3):160–176. doi:10.1016/j.jnca.2004.10.001
12. Jayatilleke GB, Padgham L, Winikoff M (2005) A model driven component-based development framework for agents. Int J Comput Syst Sci Eng 4(20):273–283
13. Khallouf J, Winikoff M (2009) Goal-oriented design of agent systems: a refinement of prometheus and its evaluation. Int J Agent-Oriented Software Eng 3(1):88–112
14. Kumar S, Huber MJ, Cohen PR (2002) Representing and executing protocols as joint actions. In: Proceedings of the first international joint conference on autonomous agents and multi-agent Systems. ACM, Bologna, pp 543–550
15. Macal CM, North MJ (2008) Agent-based modeling and simulation: ABMS examples. In: Mason SJ, Hill RR, Mönch L, Rose O, Jefferson T, Fowler JW (eds) Winter Simulation Conference, (WSC 2008) Miami, Florida. WSC, pp 101–112, 7–10 December 2008
16. Miller T, Padgham L, Thangarajah J (2010) Test coverage criteria for agent interaction testing. In: Weyns D, Gleizes MP (eds) Proceedings of the 11th International Workshop on Agent Oriented Software Engineering, pp 1–12
17. North MJ, Collier NT, Vos JR (2006) Experiences creating three implementations of the repast agent modeling toolkit. ACM Trans Model Comput Simul 16(1):1–25
18. Padgham L, Perepletchikov M (2007) Prioritisation mechanisms to support incremental development of agent systems. Int J Agent-Oriented Software Eng 1(3/4):477–497
19. Padgham L, Winikoff M (2004) Developing intelligent agent systems: a practical guide. Wiley series in agent technology. Wiley, Chichester
20. Padgham L, Winikoff M, Poutakidis D (2005) Adding debugging support to the prometheus methodology. J Eng Appl Artif Intell 18(2):173–190
21. Poutakidis D, Winikoff M, Padgham L, Zhang Z (2009) Debugging and testing of multi-agent systems using design artefacts. In: Bordini RH, Dastani M, Dix J, El Fallah Seghrouchni A (eds) Multi-agent programming: languages, tools, and applications, Chap 7. Springer, pp 215–258
22. Sierra C, Thangarajah J, Padgham L, Winikoff M (2007) Designing institutional multi-agent systems. In: Padgham L, Zambonelli F (eds) Agent Oriented Software Engineering VII: 7th International Workshop, AOSE 2006. Lecture notes in computer science. Springer, pp 84–103
23. Thangarajah J, Jayatilleke GB, Padgham L (2011) Scenarios for system requirements traceability and testing. In: Sonenberg L, Stone P, Tumer K, Yolum P (eds) 10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011), Taipei, Taiwan, vol 1–3. IFAAMAS, pp 285–292, 2–6 May 2011
24. Winikoff M (2005) JACK^{TM} Intelligent agents: an industrial strength platform. In: Bordini RH, Dastani M, Dix J, Fallah-Seghrouchni AE (eds) Multi-agent programming: languages, platforms and applications. Springer, pp 175–193

25. Winikoff M (2007) Defining syntax and providing tool support for Agent UML using a textual notation. Int J Agent-Oriented Software Eng 1(2):123–144
26. Winikoff M, Cranefield S (2008) On the testability of BDI agent systems. Information Science Discussion Paper Series 2008/03, University of Otago, Dunedin, New Zealand
27. Winikoff M, Padgham L (2013) Agent oriented software engineering. In: Weiß G (ed) Multiagent systems, Chap 15, 2nd edn. MIT Press, Cambridge, MA
28. Yolum P, Singh MP (2002) Flexible protocol specification and execution: applying event calculus planning using commitments. In: Proceedings of the 1st Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS), pp 527–534
29. Zhang Z (2011) Automated unit testing of agent systems. Ph.D. thesis, RMIT University, Melbourne, Australia
30. Zhang Z, Thangarajah J, Padgham L (2008) Automated unit testing intelligent agents in PDT. In: 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008), Demo Proceedings, Estoril, Portugal. IFAAMAS, pp 1673–1674, 12–16 May 2008
31. Zhang Z, Thangarajah J, Padgham L (2011) Automated testing for intelligent agent systems. In: Gleizes MP, Gómez-Sanz JJ (eds) Agent-Oriented Software Engineering X - 10th International Workshop, AOSE 2009, Revised Selected Papers. Lecture notes in computer science, vol 6038, Budapest, Hungary. Springer, pp 66–79, 11–12 May 2009