The Moldable Debugger: a Framework for Developing Domain-Specific Debuggers

Andrei Chiş[1], Tudor Gîrba[2], Oscar Nierstrasz[1]

[1] Software Composition Group, University of Bern, http://scg.unibe.ch
[2] CompuGroup Medical Schweiz AG, tudor@tudorgirba.com

Abstract. Debuggers are crucial tools for developing object-oriented software systems as they give developers direct access to the running systems. Nevertheless, traditional debuggers rely on generic mechanisms to explore and exhibit the execution stack and system state, while developers reason about and formulate domain-specific questions using concepts and abstractions from their application domains. This creates an abstraction gap between the debugging needs and the debugging support leading to an inefficient and error-prone debugging effort. To reduce this gap, we propose a framework for developing domain-specific debuggers called the *Moldable Debugger*. The Moldable Debugger is adapted to a domain by creating and combining *domain-specific debugging operations* with *domain-specific debugging views*, and adapts itself to a domain by selecting, at run time, appropriate debugging operations and views. We motivate the need for domain-specific debugging, identify a set of key requirements and show how our approach improves debugging by adapting the debugger to several domains.

1 Introduction

Debugging is a prerequisite for maintaining and evolving object-oriented software systems. Despite its importance it is a complex and time-consuming activity. Together with testing it can take a significant part of the effort required to build a software system [1]. Using inadequate infrastructures for performing these activities can further increase this effort [2].

Debugging is typically performed by using a debugger that allows developers to interact with a running software system and explore its state. This makes the debugger a crucial tool in any programming environment. Nevertheless, there is an abstraction gap between the way in which developers reason about object-oriented applications, and the way in which they debug them.

On the one hand, object-oriented applications use objects to capture and express a model of the application domain. Developers reason about and formulate questions using concepts and abstractions from that domain model. This fosters program comprehension as domain concepts play an important role in software development [3,4]. Furthermore, non-trivial object-oriented applications contain rich object models [5]. A common approach to improve the development and evolution of these object models is to take advantage of internal DSLs that, by making use of APIs and of the syntax of the host language, can directly express domain abstractions [6].

On the other hand, classical debuggers focusing on generic stack-based operations, line breakpoints, and generic user interfaces do not allow developers to rely on domain concepts. Approaches that address this problem by offering object-oriented debugging idioms [7] still solve only part of the problem, as they do not capture domain concepts constructed on top of object-oriented programming idioms.

Generic solutions that do not offer a one-to-one mapping between developer questions and debugging support force developers to refine their high-level questions into low-level ones and mentally piece together information from various sources. For example, when developing a parser, we might need to step through the execution until we reach a certain position in the input stream. However, as it has no knowledge of parsing and stream manipulation, a generic debugger requires us to manipulate low-level concepts like sending a message or looking up variables. This abstraction gap leads to an ineffective and error-prone effort [8].

While the debugger of a host language can be used to debug internal DSLs, it still suffers from the aforementioned limitations. When dealing with external DSLs those limitations can be addressed by automatically generating, from the grammar of the DSL, domain-specific debuggers that work at the right level of abstraction [9]. However, this solution does not apply to object-oriented applications if there is no grammar or formal specification capturing the domain model.

There exist two main approaches to address, at the application level, the gap between the debugging needs and debugging support:

- supporting domain-specific debugging operations for stepping through the execution, setting breakpoints, checking invariants [10,11,12] and querying stack-related information [13,14,15].
- providing debuggers with domain-specific user interfaces that do not necessarily have a predefined content or a fixed layout [16].

Each of these directions addresses individual debugging problems, however until now there does not exist one comprehensive approach to tackle the overall debugging puzzle. We propose an approach that incorporates both of these directions in one coherent model. We start from the realization that the most basic feature of a debugger model is to enable the customization of all aspects, and we design a debugging model around this principle. We call our approach the *Moldable Debugger*.

The Moldable Debugger decomposes a domain-specific debugger into a *domain-specific extension* and an *activation predicate*. The domain-specific extension customizes the user interface and the operations of the debugger, while the *activation predicate* captures the state of the running program in which that domain-specific extension is applicable. In a nutshell, the Moldable Debugger model allows developers to *mold* the functionality of the debugger to their own domains by creating domain-specific extensions. Then, at run time, the Moldable

Debugger adapts to the current domain by using activation predicates to select appropriate extensions.

A domain-specific extension consists of *(i)* a set of domain-specific *debugging operations* and *(ii)* a domain-specific *debugging view*, both built on top of *(iii)* a *debugging session*. The *debugging session* abstracts the low-level details of a domain. *Domain-specific operations* reify debugging operations as objects that control the execution of a program by creating and combining *debugging events*. We model debugging events as objects that encapsulate *a predicate over the state of the running program* (*e.g.*, method call, attribute mutation) [17]. A *domain-specific debugging view* consists of a set of graphical widgets that offer debugging information. Each widget locates and loads, at run-time, relevant domain-specific operations using an annotation-based approach.

To validate our model, we implemented it in Pharo[3], a modern Smalltalk environment. The Moldable Debugger implementation is written in less than 2000 lines of code. We have instantiated it for several distinct domains and each time the implementation required between 200-600 lines of code. We consider that its small size makes it easy to understand, and makes the adaptation of the debugger to specific domains an affordable activity.

The contributions of this paper are as follows:

- Identifying and discussing four requirements that an infrastructure for developing domain-specific debuggers should support;
- Presenting the Moldable Debugger, a model for creating and working with domain-specific debuggers that integrates domain-specific debugging operations with domain-specific user interfaces;
- Examples illustrating the advantages of the Moldable Debugger model over generic debuggers;
- A prototype implementation of the Moldable Debugger model.

2 Motivation

Debuggers are comprehension tools. They are often used by developers to *understand the run-time behavior of software* and *elicit run-time information* [18,19]. In test-driven development the debugger is used as a development tool given that it provides direct access to the running system [20].

Despite their importance, most debuggers only provide low-level operations that do not capture user intent and standard user interfaces that only display generic information. These issues can be addressed if developers are able to create domain-specific debuggers adapted to their problems and domains. Domain-specific debuggers can provide features at a higher level of abstraction that *(i)* match the domain model of software applications and *(ii)* group contextual information from various sources.

[3] `http://pharo.org`

In this section we establish and motivate four requirements that an infrastructure for developing domain-specific debuggers should support, namely: *domain-specific user interfaces*, *domain-specific debugging operations*, *automatic discovery* and *dynamic switching*.

2.1 Domain-specific user interfaces

User interfaces of software development tools tend to provide large quantities of information, especially as the size of systems increases. This in turn, increases the navigation effort of identifying the information relevant for a given task. While some of this effort is unavoidable, part of it is simply overhead caused by how information is organized on screen [21].

Consider a unit test with a failing equality assertion. In this case, the only information required by the developer is the difference between the expected and the actual value. However, finding the exact difference in non-trivial values can be daunting and can require multiple interactions such as finding the place in the stack where both variables are accessible, and opening separate inspectors for each values. A better approach is to show a diff view on the two values directly in the debugger when such an assertion exception occurs, without requiring any further action.

This shows that user interfaces that extract and highlight domain-specific information have the power to reduce the overall effort of code understanding [22]. However, today's debuggers tend to provide generic user interfaces that cannot emphasize what is important in application domains. To address this concern an infrastructure for developing domain-specific debuggers should:

- allow domain-specific debuggers to have *domain-specific user interfaces* displaying information relevant for their particular domains;
- support the *fast prototyping* of domain-specific user interfaces for debugging.

While other approaches, like *deet* [23] and *Debugger Canvas* [16], support domain-specific user interfaces for different domains, they do not offer an easy and rapid way to develop such domain-specific user interfaces.

2.2 Domain-specific debugging operations

Debugging is viewed as a laborious activity requiring much manual and repetitive work. On the one hand, debuggers support language-level operations. As a consequence, developers need to mentally construct high-level abstractions on top of them, which can be time-consuming. On the other hand, debuggers rarely provide support for identifying and navigating through those high-level abstractions. This leads to repetitive tasks that increase debugging time.

Consider a framework for synchronous message passing. One common use case in applications using it is the delivery of a message to a list of subscribers. When debugging this use case, a developer might need to *step to when the current message is delivered to the next subscriber*. One solution is to manually step through the execution until the desired code location is reached. Another consists in identifying the code location beforehand, setting a breakpoint there and resuming execution. In both cases developers have to manually perform a series of actions each time they want to execute this high-level operation.

A predefined set of debugging operations cannot anticipate and capture all relevant situations. Furthermore, depending on the domain different debugging operations are of interest. Thus, an infrastructure for developing domain-specific debuggers should:

- *support the creation of domain-specific debugging operations* that allow developers to *express and automate* high-level abstractions from application domains (*e.g.*, creating domain-specific breakpoints, building and checking invariants, altering the state of the running system). Since developers view debugging as an event-oriented process, the underlying mechanism should allow developers to treat the running program as a generator of events, where an event corresponds to the occurrence of a particular action during the program's execution, like: method entry, attribute access, attribute write, memory access, *etc.*
- group together those debugging operations that are relevant for a domain and only make them available to developers when they encounter that domain.

This idea of having *customizable* or *programmable* debugging operations that view debugging as an event-oriented activity has been supported in related works [10,11,12,23]. Mainstream debuggers like GDB have, to some extent, also incorporated it. We also consider that debugging operations should be grouped based on the domain and only usable when working with that domain.

2.3 Automatic discovery

Based on an observational study of 28 professional developers *Roehm et al.* report that none of them used a dedicated program comprehension tool; some were not aware of standard features provided by their IDE [18]. Another study revealed that despite their usefulness and long lasting presence in IDEs, refactoring tools are heavily underused [24].

In the same way, *developers need help to discover domain-specific debuggers during debugging.* For example, if while stepping through the execution of a program a developer reaches a parser, the developer should be informed that a domain-specific debugger exists that can be used in that context; if later the execution of the parser completes and the program continues with the propagation of an event, the developer should be informed that the current domain-specific debugger is no longer useful and that a better one exists. This way, the burden of finding appropriate domain-specific debuggers and determining when they are applicable does not fall on developers.

Recommender systems typically address the problem of locating useful software tools/commands by recording and mining usage histories of software tools [25] (*i.e.*, what tools developers used as well as how they used them). This requires, at least, some usage history information. To eliminate this need an infrastructure for developing domain-specific debugger should *allow each domain-specific debugger to encapsulate the situations/domains in which it is applicable.*

2.4 Dynamic switching

Even with just two different types of debuggers, *DeLine et al.* noticed that users needed to switch between them at run time [16]. This happened as users did not know in advance in what situation they would find themselves in during debugging. Thus, they often did not start with the appropriate one.

Furthermore, even if one starts with the right domain-specific debugger, during debugging situations can arise requiring a different one. For example, the following scenario can occur: *(i)* while investigating how an event is propagated through the application *(ii)* a developer discovers that it is used to trigger a script constructing a GUI, and later learns that *(iii)* the script uses a parser to read the content of a file and populate the GUI. At each step a different domain-specific debugger can be used. For this to be feasible, *domain-specific debuggers should be switchable at debug time without having to restart the application.*

2.5 Summary

Generic debuggers focusing on low-level programming constructs, while universally applicable, cannot efficiently answer domain-specific questions, as they make it difficult for developers to take advantage of domain concepts. Domain-specific debuggers aware of the application domain can provide direct answers. We advocate that a debugging infrastructures for developing domain-specific debuggers should support the four aforementioned requirements (*domain-specific user interfaces*, *domain-specific debugging operations*, *automatic discovery* and *dynamic switching*).

3 Introducing the "Moldable Debugger" model

Conventional debuggers force developers to use generic constructs to address domain-specific problems. The Moldable Debugger, on the other hand, explicitly supports domain-specific debuggers that can express and answer questions at the application level. A domain-specific debugger consists of *a domain-specific extension* encapsulating the functionality and *an activation predicate* encapsulating the situations in which the extension is applicable. This model makes it possible for multiple domain-specific debuggers to coexist at the same time.

To exemplify the ideas behind the proposed solution we will instantiate a domain-specific debugger for working with synchronous events[4]. Event-based programming poses debugging challenges as it favors a control flow based on events not supported well by conventional stack-based debuggers.

[4] This section briefly describes this debugger. More details are given in Section 4.2.

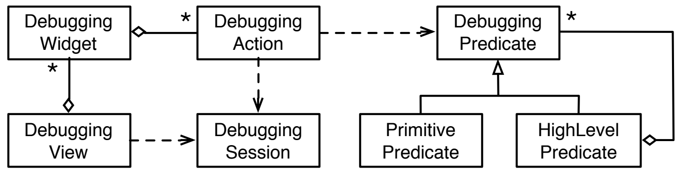

Fig. 1: The structure of a domain-specific extension.

3.1 Modeling domain-specific extensions

A domain-specific extension defines the functionality of a domain-specific debugger using multiple *debugging operations* and a *debugging view*. Debugging operations rely on *debugging predicates* to implement high-level abstractions (*e.g.*, domain-specific breakpoints); the debugging view highlights contextual information. To decouple these components from the low-level details of a domain they are built on top of a *debugging session*.

A *debugging session* encapsulates the logic for working with processes and execution contexts (*i.e.*, stack frames). It further implements common stack-based operations like: *step into*, *step over*, *resume/restart process*, *etc.* Domain-specific debuggers can extend the debugging session to extract and store custom information from the runtime, or provide fine-grained debugging operations. For example, our event-based debugger extends the debugging session to extract and store the current event together with the sender and the receiver of that event.

Debugging predicates detect *run-time events*. Basic run-time events (*e.g.*, method call, attribute access) are detected using a set of *primitive predicates*, detailed in Table 1. More complex run-time events are detected using *high-level predicates* that combine both *primitive predicates* and other *high-level predicates* (Figure 1). Both these types of debugging predicates are encapsulated as objects whose state does not change after creation.

Consider our event-based debugger. This debugger can provide high-level predicates to detect when a sender initiates the delivery of an event, or when the middleware delivers the event to a receiver.

Attribute read	detects when a field of any object of a certain type is accessed
Attribute write	detects when a field of any object of a certain type is mutated
Method call	detects when a given method is called on any object of a certain type
Message send	detects when a specified method is invoked from a given method
State check	checks a generic condition on the state of the running program (*e.g.*, the identity of an object).

Table 1: Primitive debugging predicates capturing basic events.

Debugging operations can execute the program until a debugging predicate is matched or can perform an action every time a debugging predicate is matched. They are modeled as objects that can accumulate state. They can implement breakpoints, log data, watch fields, change the program's state, detect violations of invariants, *etc.* In the previous example a debugging operation can be used to stop the execution when an event is delivered to a receiver. Another debugging operation can log all events delivered to a particular receiver without stopping the execution. At each point during the execution of a program only a single debugging operation can be active. Thus, debugging operations have to be run sequentially. This design decision simplifies the implementation of the model, given that two conflicting operations cannot run at the same time.

The Moldable Debugger models a *debugging view* as a collection of graphical widgets (*e.g.*, stack, code editor, object inspector) arranged using a particular layout. At run time, each widget loads a subset of debugging operations. Determining what operations are loaded by which widgets is done at run time via a lookup mechanism of operation declarations (implemented in practice using annotations). This way, widgets do not depend upon debugging operations, and are able to reload debugging operations dynamically during execution.

Our event-based debugger provides dedicated widgets that display an event together with the sender and the receiver of that event. These widgets load and display the debugging operations for working with synchronous events, like logging all events or placing a breakpoint when an event is delivered to a receiver.

Developers can create domain-specific extensions by:

(i) extending the debugging session with additional functionality;
(ii) creating domain-specific debugging predicates and operations;
(iii) specifying a domain-specific debugging view;
(iv) linking debugging operations to graphical widgets;

3.2 Dynamic Integration

The Moldable Debugger model enables each domain-specific debugger to decide if it can handle or not a debugging situation by defining an *activation predicate*. Activation predicates capture the state of the running program in which a domain-specific debugger is applicable. While debugging predicates are applied on an execution context, activation predicates are applied on the entire execution stack. For example, the activation predicate of our event-based debugger will check if the execution stack contains an execution context involving an event.

This way, developers do not have to be aware of applicable debuggers a priori. At each point during debugging they can see what domain-specific debuggers are applicable (*i.e.*, their activation predicate matches the current debugging context) and can switch to any of them.

When a domain-specific debugger is no longer appropriate we do not automatically switch to another one. Instead, all domain-specific widgets and operations are disabled. This avoids confronting users with unexpected changes in the user interface if the new debugging view has a radically different layout/content.

To further improve working with multiple domain-specific debuggers we provide two additional concepts:

(i) *A debugger-oriented breakpoint* is a breakpoint that when reached opens the domain-specific debugger best suited for the current situation. If more than one view is available the developer is asked to choose one.

(ii) *Debugger-oriented steps* are debugging operations that resume execution until a given domain-specific debugger is applicable. They are useful when a developer knows a domain-specific debugger will be used at some point in the future, but is not sure when or where.

4 Addressing domain-specific debugging problems

To demonstrate that the Moldable Debugger addresses the requirements identified in Section 2 we have instantiated it for four applications belonging to different domains: *testing*, *synchronous events*, *parsing* and *internal DSLs*. In this section we detail these instantiations.

4.1 Testing with SUnit

SUnit is a framework for creating unit tests [26]. The framework provides an assertion to check if a computation results in an expected value. If the assertion fails the developer is presented with a debugger that can be used to compare the obtained value with the expected one. If these values are complex, identifying the difference may be time consuming. A solution is needed to *facilitate comparison*.

To address this, we developed a domain-specific debugger having the following components:

Session: extracts the expected and the obtained value from the runtime;
View: displays a diff between the textual representation of the two values. The diff view depends on the domain of the data being compared.
Activation predicate: verifies if the execution stack contains a failing equality assertion.

4.2 An Announcement-Centric debugger

The *Announcements* framework from Pharo provides a synchronous notification mechanism between objects based on a registration mechanism and first class announcements (*i.e.*, objects storing all information relevant to particular occurrences of events). Since the control flow for announcements is event-based, it does not match well the stack-based paradigm used by conventional debuggers. For example, Section 2.2 describes a high-level action for *delivering an announcement to a list of subscribers*. Furthermore, when debugging announcements it is useful *to see at the same time both the sender and the receiver of an announcement*; most debuggers only show the receiver.

To address these problems we have created a domain-specific debugger, shown in Figure 2. A previous work discusses in more details the need for such a debugger and looks more closely at the runtime support needed to make the debugger possible [27]. This debugger is instantiated as follows:

Session: extracts from the runtime the announcement, the sender, the receiver and all the other subscriptions triggered by the current announcement;

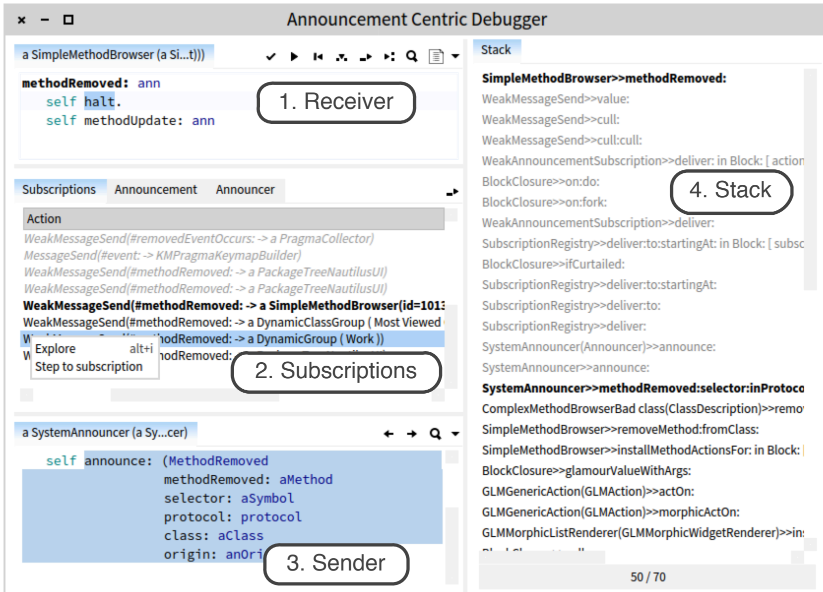

Fig. 2: A domain-specific debugger for announcements: (1)(3) the receiver and the sender of an announcement; (2) subscriptions triggered by the current announcement.

Predicates: (i) detect when the framework initiates the delivery of a subscription; (ii) detect when the framework delivers a subscription to an object;

Operations: (i) step to the delivery of the next subscription; (ii) step to the delivery of a selected subscription;

View: shows both the sender and the receiver of an announcement, together with all subscriptions served as a result of that announcement;

Activation predicate: verifies if the execution stack contains an execution context involving an announcement.

4.3 A debugger for PetitParser

PetitParser is a framework for creating parsers, written in Pharo, that makes it easy to dynamically reuse, compose, transform and extend grammars [28]. A parser is created by specifying a set of grammar productions in one or more dedicated classes. When a parser is instantiated the grammar productions are used to create a tree of primitive parsers (*e.g.*, choice, sequence, negation, *etc.*); this tree is then used to parse the input.

Whereas most parser generators instantiate a parser by generating code, PetitParser generates a dynamic graph of objects. Nevertheless, the same issues arise as with conventional parser generators: generic debuggers do not provide debugging operations at the level of the input (*e.g.*, set a breakpoint when a certain part of the input is parsed) and of the grammar (*e.g.*, set a breakpoint when a grammar production is exercised). Generic debuggers also do not display the source code of grammar productions nor do they provide easy access to the input being parsed.

We have developed a domain-specific debugger for PetitParser by configuring the Moldable Debugger as follows:

Session: extracts from the runtime the parser and the input being parsed;

Predicates: (i) detect the usage of a primitive parser; (ii) detect the usage of a production; (iii) detect when a parser fails to match the input; (iv) detect when the position of the input stream changes to a given value;

Operations: Navigating through the execution at a higher level of abstraction is supported through the following debugging operations:
- *Next parser*: step until a primitive parser of any type is reached
- *Next production*: step until a production is reached
- *Production(aProduction)*: step until the given production is reached
- *Next failure*: step until a parser fails to match the input
- *Stream position change*: step until the stream position changes (it either increases, if a character was parsed, or decrease if the parser backtracks)
- *Stream position(anInteger)*: step until the stream reaches a given position

View: The debugging view of the resulting debugger is shown in Figure 3. We can see that now the input being parsed is incorporated into the user interface; to know how much parsing has advanced, the portion that has already been parsed is highlighted. Tabs are used to group six widgets showing different types of data about the current production, like: source code, structure, position in the whole graph of parsers, an example that can be parsed with the production, *etc.* The execution stack further highlights those execution contexts that represent a grammar production;

Activation predicate: verifies if the execution stack contains an execution context created when using a parser.

4.4 A debugger for Glamour

Glamour is an engine for scripting browsers based on a components and connectors architecture [29]. New browsers are created by using an internal domain-specific language (DSL) to specify a set of *presentations* (graphical widgets) along with a set of *transmissions* between those presentations, encoding the information flow. Users can attach various conditions to transmissions and alter the information that they propagate. Presentations and transmissions form a model that is then used to generate the actual browser.

The Moldable Debugger relies on Glamour for creating domain-specific views. Thus, during the development of the framework we created a domain-specific debugger to help us understand the creation of a browser:

Session: extracts from the runtime the model of the browser;

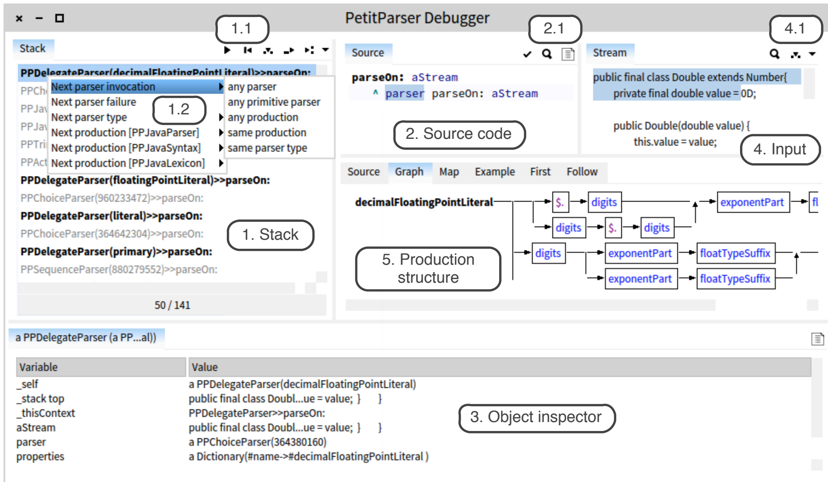

Fig. 3: A domain-specific debugger for PetitParser. The debugging view displays relevant information for debugging parsers ((4) Input, (5) Production structure). Each widget loads relevant debugging operations (1.1, 1.2, 2.1, 4.1).

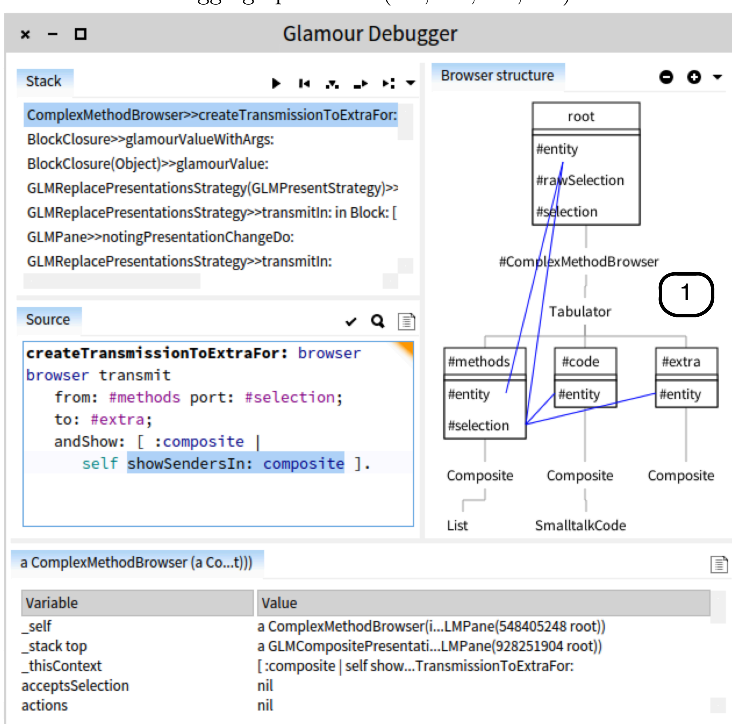

Fig. 4: A domain-specific debugger for Glamour: (1) visualization showing the model of the browser currently constructed.

Predicates: (i) detect the creation of a presentation; (ii) detect when a transmission alters the value that it propagates; (iii) detect when the condition of a transmission is checked;
Operations: (i) step to presentation creation; (ii) step to transmission transformation; (iii) step to transmission condition;
View: displays the structure of the model in an interactive visualization that is updated as the construction of the model advances (Figure 4);
Activation predicate: verifies if the execution stack contains an execution context that triggers the construction of a browser.

4.5 Summary

PetitParser, Glamour, SUnit and the Announcements framework cover four distinct domains. For each one we were able to instantiate a domain-specific debugger having a contextual debugging view and/or a set of debugging operations capturing high-level abstractions from that domain. This shows the Moldable Debugger framework addresses the first two requirements.

The two remaining requirements, *automatic discovery* and *dynamic switching*, are also addressed. At each point during debugging developers can obtain a list of all domain-specific debuggers applicable to their current context. This does not require them either to know in advance all available debuggers, or to know when those debuggers are applicable. Once the right debugger was found developers can switch to it and continue debugging without having to restart the application. For example, one can perform the scenario presented in Section 2.4. The cost of creating these debuggers is discussed in Section 6.1.

5 Implementation

The current prototype of the Moldable Debugger[5] is implemented in Pharo, an open-source Smalltalk inspired environment. In this section we discuss several aspects regarding its implementation.

5.1 Controlling the execution

In the current version the target program is controlled based on debugging predicates that are checked in a *step-by-step* manner after executing each instruction [30,31]. To do this we transform each debugging predicate into a boolean condition that is applied on the execution context. For example, the debugging predicate for detecting if a parser has failed forms a boolean condition that checks if an execution context was created as a result of sending the message initializeMessageAt to an instance of the class PPFailure.

[5] More details including demos and installation instructions can be found at: `http://scg.unibe.ch/research/moldabledebugger`

The main advantage of this method is that it is simple to understand and it does not alter the source of the target program. However, it can slow down the target program considerably. To address this concern, debugging operations do not have to be aware that predicates are used to control the target program in a *step-by-step* manner. Thus, a backend based on a different approach, like code instrumentation, could be used. We are currently looking at how to instrument code based on predicates. For example, the previous predicate could be used to instrument the method initializeMessageAt of the class PPFailure.

These two views of either using boolean conditions or code instrumentation to implement debugging operations match the *step* and *break* constructs proposed by *Crawford et al.* [30]. As they discuss, their combination can lead to semantic issues. To avoid those issues only a debugging operation can be active at a time, and debugging operations should not combine instrumentation with step-by-step execution.

5.2 The Moldable Debugger in other languages

The current prototype of the Moldable Debugger is implemented in Pharo. It can be ported to other languages as long as they provide a good infrastructure for controlling the execution of a target program and there exists a way to rapidly construct user interfaces for debuggers.

For example, one could implement the framework in Java. Domain-specific debugging operations can be implemented on top of the Java Debugging Interface (JDI) or by using aspects. JDI is a good candidate as it provides explicit control over the execution of a virtual machine and introspective access to its state. Aspect-Oriented Programming [32] can implement debugging actions by instrumenting only the code locations of interest. Dynamic aspects (*e.g.*, AspectWerkz [33]) can further scope code instrumentation at the debugger level. Last but not least, domain-specific views can be obtained by leveraging the functionality of IDEs, like *perspectives* in the Eclipse IDE.

6 Discussion

6.1 The cost of creating new debuggers

The four presented domain-specific debuggers were created starting from a model consisting of 1500 lines of code. Table 2 shows, for each debugger, how many lines of code were needed for the debugging view, the debugging actions, and the debugging session.

Even if, in general, *lines of code* (LOC) must be considered with caution when measuring complexity and development effort, as the metric does not indicate the time needed to write those lines, it gives a good indication of the small size of these domain-specific debuggers. This small size makes the construction cost affordable. Similar conclusions can be derived from the work of *Kosar et al.* that shows that with the right setup its possible to construct a domain-specific debugger for a modelling language with relatively low costs [34].

	Session	Operations	View	Total
Base model	800	700	-	1500
Default Debugger	-	100	400	500
Announcements	200	50	200	450
Petit Parser	100	300	200	600
Glamour	150	100	50	300
SUnit	100	-	50	150

Table 2: Size of extensions in lines of code (LOC).

The availability of such an infrastructure opens new possibilities:

(i) the developers of a library or framework can create and ship a dedicated debugger together with the code, to help users debug that framework or library. For example, we can envisage the developers of PetitParser and Glamour to have built the custom debuggers themselves and ship them together with the frameworks;

(ii) developers can extend the debugger for their own applications, during the development process, to help them solve bugs or better understand the application.

6.2 IDE Integration

Studies of software developers revealed that they use standalone tools alongside an IDE, even when their IDE has the required features [18]. Furthermore, developers also complain about loose integration of tools that forces them to look for relevant information in multiple places [35]. To avoid these problems the Moldable Debugger framework is integrated into the Pharo IDE and essentially replaces the existing debugger.

The Moldable Debugger along with the domain-specific debuggers presented in Section 4 are also integrated into Moose[6], a platform for data and software analysis [36]. Despite the fact that the performance of the current implementation can be significantly improved, these domain-specific debuggers are usable and reduce debugging time. For example, we are using the domain-specific debugger for PetitParser on a daily basis.

6.3 Open questions

As software systems evolve domain-specific debuggers written for those systems must also evolve. This raises further research questions like: *"What changes in the application will lead to changes in the debugger?"* or *"How can the debugger be kept in sync with the application?"*. For example, introducing code instrumentation or destructive data reading (as in a stream) can lead to significant changes in an existent debugger.

[6] `http://moosetechnology.org`

In this context, a more high-level question is *"What makes an application debuggable?"*. By this we mean what characteristics of an application ease, or exacerbate the creation of debuggers or, more generally, what characteristics affect debugging. To draw an analogy, in the field of software testing high coupling makes the creation of unit tests difficult (by increasing the number of dependencies that need to be taken into account) and thus decreases the testability of a software system.

7 Related Work

This work draws its ideas from programmable/scriptable debugging and debugging infrastructures for language workbenches. For clarity we discuss related work with respect to how other approaches support domain-specific debugging operations and user-interfaces for debugging.

7.1 Specifying domain-specific operations

There is a wide body of research on allowing developers to automate debugging tasks by creating high-level abstractions. *MzTake* [11] is a scriptable debugger allowing developers to automate debugging tasks. It treats a running program as a stream of events that can be analyzed using operators, like *map* and *filter*; streams can also be combined to form new streams. The focus in *MzTake* is on automating debugging actions using scripts. It does not provide support for creating domain-specific views for debugging. Developers just have the possibility of visually exploring data by using features from the host IDE, DrScheme.

Dalek [10] is a C debugger employing a dataflow approach for debugging sequential programs: developers create high-level events by combining different types of low-level events. *Coca* [37] is an automated debugger for C using Prolog predicates to search for events of interest over program state. *Acid* [38] makes it possible to write debugging operations, like breakpoints and step instructions, in a language designed for debugging that reifies program state as variables. *Duel* [39] is a high-level language on top of GDB for writing state exploration queries. *Expositor* [12] is a scriptable time-travel debugger that can check temporal properties of an execution: it views program traces as immutable lists of time-annotated program state snapshots and uses an efficient data structure to manage them. These approaches focus on improving debugging by allowing developers to create different types of commands, breakpoints or queries at a higher level of abstraction. However, they have the same drawbacks as *MzTake*: by focusing only on operations they neglect the user interface of debuggers. They also do not provide support for selecting features based on the debugging context.

Object-centric debugging [7] proposes a new way to perform debugging operations by focusing on objects instead of the execution stack. *Reverse watchpoints* use the concept of *position* to automatically find the last time a target variable was written and move control flow to that point [40]. *Whyline* is a debugging tool that allows developer to ask and answer *Why* and *Why Not* questions about program behavior [41]. *Query-based debugging* facilitates the creation of queries over program execution and state using high-level languages [13,14,15]. These approaches are complementary to our approach as they can be used to create other types of debugging operations.

Language workbenches for domain-specific languages (DSL) address debugging by offering debugging abstractions at the level of the DSL [9,42,43]. This solves the debugging problem both at the language and application level only if domain concepts are incorporated directly into the DSL. However, if domain concepts are build on top of a DSL, then DSL debuggers suffer from the same limitations as generic debuggers. Our approach supports, in all cases, debuggers aware of application domains.

7.2 User interfaces for debugging

Debugger Canvas [16] proposes a novel type of user interface for debuggers based on the *Code Bubbles* [44] paradigm. Rather then starting from a user interface having a predefined structure, developers start from an empty one on which different bubbles are added, as they step through the execution of the program. Our approach allows developers to create custom user interfaces (views) beforehand and select appropriate interfaces at debug time. *Debugger Canvas* focuses only on the user interface, and does not provide support for adding custom debugging operations. Our approach addresses both aspects.

The Data Display Debugger (DDD) [45] is a graphical user interface for GDB providing a graphical display for representing complex data structures as graphs that can be explored incrementally and interactively. However, if focuses just on providing a default front-end for GDB; it does not offer support for customization, nor other debugging operations then the ones provided by GDB.

jGRASP supports the visualization of various data structure by means of dynamic viewers and a *structure identifier* that automatically select suitable views for data structures [46]. xDIVA is a 3-D debugging visualization system where complex visualization metaphors are assembled from individual ones, each of which is independently replaceable [47]. While these approaches allow users to create visualizations specific to their domains they are meant to be embedded within existent debuggers, and thus do not offer debugging operations.

7.3 Unifying approaches

deet [23] is a debugger for ANSI C that, like our approach, promotes simple debuggers having few lines of code. It further allows developers to extend the user interface and add new commands by writing code in a high-level language. *TIDE* is a debugging framework focusing on the instantiation of debuggers for formal languages (ASF+SDF, in particular) [48]; developers can implement high-level debugging actions like, breakpoints and watchpoints, extend the user interface be modifying the Java implementation of TIDE, and use *debugging rules* to state which debugging actions are available at which logical breakpoints. Unlike these approaches, we propose modeling the customization of debugger through explicit domain-specific extensions and provide support for automatically detecting appropriate extensions at run time.

LISA is a grammar-based compiler generator that can automatically generate debuggers, inspectors and visualizers for DSLs that have a formal language specification [49]. Our approach targets object-oriented systems where such a formal specification is missing.

8 Conclusions

Developers encounter domain-specific questions. Traditional debuggers supporting debugging by means of generic mechanisms, while universally applicable, are less suitable to handle domain-specific questions. The Moldable Debugger addresses this contradiction by allowing developers to created domain-specific debuggers having both custom debugging actions and user interfaces, with a low effort. As a validation, we implemented the Moldable Debugger model and created four different debuggers in less than 600 lines of code each. The Moldable Debugger reduces the abstraction gap between the debugging needs and debugging support leading to a more efficient and less error-prone debugging effort.

Given the large costs associated with debugging activities, improving the workflow and reducing the cognitive load of debugging can have a significant practical impact. With our approach developers can create their own debuggers to address recurring custom problems. This can make considerable economical sense when working on a long lived system. Furthermore, library developers can ship library-specific debuggers together with their product. This can have a practical impact due to the reuse of the library in many applications.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assessment" (SNSF project Nr. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015). We thank Alexandre Bergel, Jorge Ressia, and the anonymous reviewers for their suggestions in improving this paper. We also thank CHOOSE, the special interest group for Object-Oriented Systems and Environments of the Swiss Informatics Society, for its financial contribution to the presentation of this paper.

References

1. Vessey, I.: Expertise in Debugging Computer Programs: An Analysis of the Content of Verbal Protocols. IEEE Trans. on Systems, Man, and Cybernetics **16**(5) (September 1986) 621–637
2. Tassey, G.: The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology (2002)

3. Littman, D.C., Pinto, J., Letovsky, S., Soloway, E.: Mental models and software maintenance. Journal of Systems and Software **7**(4) (December 1987) 341–355
4. Rajlich, V., Wilde, N.: The role of concepts in program comprehension. In: Proc. IWPC. (2002) 271–278
5. Renggli, L., Gîrba, T., Nierstrasz, O.: Embedding languages without breaking tools. In: Proc. of ECOOP. (2010) 380–404
6. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional (2010)
7. Ressia, J., Bergel, A., Nierstrasz, O.: Object-centric debugging. In: Proc. ICSE. (2012) 485–495
8. Sillito, J., Murphy, G.C., De Volder, K.: Asking and answering questions during a programming change task. IEEE Trans. Softw. Eng. **34** (July 2008) 434–451
9. Wu, H., Gray, J., Mernik, M.: Grammar-driven generation of domain-specific language debuggers. Softw. Pract. Exper. **38**(10) (August 2008) 1073–1103
10. Olsson, R.A., Crawford, R.H., Ho, W.W.: A dataflow approach to event-based debugging. Software - Practice and Experience **21**(2) (February 1991) 209–229
11. Marceau, G., Cooper, G.H., Spiro, J.P., Krishnamurthi, S., Reiss, S.P.: The design and implementation of a dataflow language for scriptable debugging. Automated Software Engineering **14**(1) (March 2007) 59–86
12. Khoo, Y.P., Foster, J.S., Hicks, M.: Expositor: scriptable time-travel debugging with first-class traces. In: Proc. ICSE. (2013) 352–361
13. Lencevicius, R., Hölzle, U., Singh, A.K.: Query-based debugging of object-oriented programs. In: Proc. OOPSLA. (1997) 304–317
14. Potanin, A., Noble, J., Biddle, R.: Snapshot query-based debugging. In: Proc. ASWEC. (2004) 251
15. Martin, M., Livshits, B., Lam, M.S.: Finding application errors and security flaws using PQL: a program query language. In: Proc. OOPSLA, ACM (2005) 363–385
16. DeLine, R., Bragdon, A., Rowan, K., Jacobsen, J., Reiss, S.P.: Debugger canvas: industrial experience with the code bubbles paradigm. In: ICSE. (2012) 1064–1073
17. Auguston, M., Jeffery, C., Underwood, S.: A framework for automatic debugging. In: Proc. ASE'02, IEEE Computer Society (2002) 217–222
18. Roehm, T., Tiarks, R., Koschke, R., Maalej, W.: How do professional developers comprehend software? In: Proc. ICSE. (2012) 255–265
19. Murphy, G.C., Kersten, M., Findlater, L.: How are Java software developers using the Eclipse IDE? IEEE Software (July 2006)
20. Beck, K.: Test Driven Development: By Example. Addison-Wesley Longman (2002)
21. Ko, A., Myers, B., Coblenz, M., Aung, H.: An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Trans. Softw. Eng. **32**(12) (December 2006) 971 –987
22. Kersten, M., Murphy, G.C.: Mylar: a degree-of-interest model for IDEs. In: Proc. AOSD. (2005) 159–168
23. Hanson, D.R., Korn, J.L.: A simple and extensible graphical debugger. In: WINTER 1997 USENIX CONFERENCE. (1997) 173–184
24. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. In: Proc. ICSE. (2009) 287–297
25. Murphy-Hill, E., Jiresal, R., Murphy, G.C.: Improving software developers' fluency by recommending development environment commands. In: FSE. (2012) 42:1–42:11
26. Beck, K.: Kent Beck's Guide to Better Smalltalk. Sigs Books (1999)
27. Chis, A., Nierstrasz, O., Gîrba, T.: Towards a moldable debugger. In: Proc. DYLA. (2013) 2:1–2:6
28. Renggli, L., Ducasse, S., Gîrba, T., Nierstrasz, O.: Practical dynamic grammars for dynamic languages. In: Proc. DYLA. (2010)

29. Bunge, P.: Scripting browsers with Glamour. Master's thesis, University of Bern (2009)
30. Crawford, R.H., Olsson, R.A., Ho, W.W., Wee, C.E.: Semantic issues in the design of languages for debugging. Comput. Lang. **21**(1) (April 1995) 17–37
31. Lieberman, H., Fry, C.: ZStep 95: A reversible, animated source code stepper. MIT Press (1998)
32. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Aksit, M., Matsuoka, S., eds.: Proc. of ECOOP'97. Volume 1241 of LNCS., Springer-Verlag (June 1997) 220–242
33. Bonér, J.: What are the key issues for commercial AOP use: how does AspectWerkz address them? In: Proc. AOSD. (2004) 5–6
34. Kosar, T., Mernik, M., Gray, J., Kos, T.: Debugging measurement systems using a domain-specific modeling language. Computers in Industry **65**(4) (2014) 622 – 635
35. Maalej, W.: Task-First or Context-First? Tool Integration Revisited. In: Proc. ASE. (2009) 344–355
36. Nierstrasz, O., Ducasse, S., Gîrba, T.: The story of Moose: an agile reengineering environment. In: Proc. of ESEC/FSE. (2005) 1–10 Invited paper.
37. Ducassé, M.: Coca: An automated debugger for C. In: International Conference on Software Engineering. (1999) 154–168
38. Winterbottom, P.: ACID: A debugger built from a language. In: USENIX Technical Conference. (1994) 211–222
39. Golan, M., Hanson, D.R.: Duel — a very high-level debugging language. In: USENIX Winter. (1993) 107–118
40. Maruyama, K., Terada, M.: Debugging with reverse watchpoint. In: QSIC. (2003)
41. Ko, A.J., Myers, B.A.: Debugging reinvented: Asking and answering why and why not questions about program behavior. In: Proc. of ICSE. (2008) 301–310
42. Lindeman, R.T., Kats, L.C., Visser, E.: Declaratively defining domain-specific language debuggers. In: Proc. GPCE. (2011) 127–136
43. Kolomvatsos, K., Valkanas, G., Hadjiefthymiades, S.: Debugging applications created by a domain specific language: The IPAC case. J. Syst. Softw. **85**(4) (April 2012) 932–943
44. Bragdon, A., Zeleznik, R., Reiss, S.P., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., LaViola, Jr., J.J.: Code bubbles: a working set-based interface for code understanding and maintenance. In: CHI. (2010) 2503–2512
45. Zeller, A., Lütkehaus, D.: DDD — a free graphical front-end for Unix debuggers. SIGPLAN Not. **31**(1) (1996) 22–27
46. Cross, II, J.H., Hendrix, T.D., Umphress, D.A., Barowski, L.A., Jain, J., Montgomery, L.N.: Robust generation of dynamic data structure visualizations with multiple interaction approaches. Trans. Comput. Educ. **9**(2) (June 2009) 13:1–13:32
47. Cheng, Y.P., Chen, J.F., Chiu, M.C., Lai, N.W., Tseng, C.C.: xDIVA: a debugging visualization system with composable visualization metaphors. OOPSLA Companion (2008) 807–810
48. van den Brand, M.G.J., Cornelissen, B., Olivier, P.A., Vinju, J.J.: TIDE: A generic debugging framework — tool demonstration —. Electron. Notes Theor. Comput. Sci. **141**(4) (December 2005) 161–165
49. Henriques, P.R., Pereira, M.J.V., Mernik, M., Lenic, M., Gray, J., Wu, H.: Automatic generation of language-based tools using the LISA system. IEE Software Journal **152**(2) (2005) 54–69