An Extensible Framework for Variable-Precision Data-Flow Analyses in MPS

Tamás Szabó
itemis, Germany /
Delft University of Technology,
Netherlands
tamas.szabo@itemis.de

Simon Alperovich
JetBrains,
Czech Republic
alpsm@yandex.ru

Markus Voelter
independent / itemis,
Germany
voelter@acm.org

Sebastian Erdweg
Delft University of Technology,
Netherlands
s.t.erdweg@tudelft.nl

ABSTRACT

Data-flow analyses are used as part of many software engineering tasks: they are the foundations of program understanding, refactorings and optimized code generation. Similar to general-purpose languages (GPLs), state-of-the-art domain-specific languages (DSLs) also require sophisticated data-flow analyses. However, as a consequence of the different economies of DSL development and their typically relatively fast evolution, the effort for developing and evolving such analyses must be lowered compared to GPLs. This tension can be resolved with dedicated support for data-flow analyses in language workbenches.

In this tool paper we present MPS-DF, which is the component in the MPS language workbench that supports the definition of data-flow analyses for DSLs. Language developers can define data-flow graph builders declaratively as part of a language definition and compute analysis results efficiently based on these data-flow graphs. MPS-DF is extensible such that it does not compromise the support for language composition in MPS. Additionally, clients of MPS-DF analyses can run the analyses with variable precision thus trading off precision for performance. This allows clients to tailor an analysis to a particular use case.

Demo video of MPS-DF: https://youtu.be/laNDAZCe2jM.

CCS Concepts

•**Software and its engineering** → **Automated static analysis; Data flow languages;** *Integrated and visual development environments;*

Keywords

Data-flow Analysis; Domain-specific Language; Language Workbench, Inter-procedural Analysis

1. INTRODUCTION

Data-flow analysis is essential for program analysis, optimizations in compilers, integrated development environments (IDEs) and debuggers. Fast and precise analyzers are typically designed specifically for a particular programming language and require significant research and implementation effort [5, 6, 15]. This effort is justifiable for GPLs, which change rather slowly and where the implementation effort for an analysis (and other compiler and IDE features) is justified by a large user base.

Historically, most DSLs have been simple languages, requiring only simple analyses (e.g., enforcement of name uniqueness, detecting simple bug patterns or basic type checking). However, for state-of-the-art DSLs, such as mbeddr [19], WebDSL [17] or Polar [12], this is no longer true: sophisticated analyses, similar to those in GPLs, are required.

However, the economies of DSL development are different: DSLs evolve more rapidly and their user base is smaller, making it hard to justify high development effort for languages and tools in general and for analyses in particular. To lower this effort, we propose that language workbenches [8] directly support the development of data-flow analyses. This allows for the analyses to be an integral part of language development and to evolve together with the language as it grows in complexity.

In this tool paper we present MPS-DF, the *data-flow support* of the Meta Programming System (MPS).[1] MPS is a language workbench for the definition of (domain-specific) languages plus their accompanying IDEs. MPS-DF is a component of MPS, and it supports the definition and efficient execution of data-flow analyses. Users of MPS-DF first define *data-flow builders* for the analyzed language. These builders contribute subgraphs to the *data-flow graph (DFG)*, an intermediate program representation encoding the data-flow of the analyzed program. MPS-DF then supports defining *data-flow analyses*, on the DFG, which compute some data-flow specific knowledge (e.g. which variables are initialized) about the program. These data-flow analyses are static program analyses, which derive the knowledge without actually running the analyzed program. Finally, existing MPS components, such as program validators, transformations or refactorings, make use of this knowledge.

[1] https://www.jetbrains.com/mps

Contributions MPS-DF has two important characteristics: extensibility and variable precision. *Extensibility* is motivated by the fact that MPS-based languages are extensible themselves (wrt. syntax, semantics, IDE support) [7]. It comes in two flavours: first, builders enable extensibility of the DFG in the face of language extensions of the analyzed language. This means that an existing analysis immediately works on an extended program if the concepts in the language extension also define builders and thus contribute subgraphs to the DFG. The second flavour of extensibility supports augmenting the DFG for a *particular* analysis with custom nodes that encode specific knowledge about the analyzed program and potentially override an analysis result based on that knowledge.

Variable precision considers performance trade-offs: real-time checks in the IDE must run fast, possibly compromising on precision, whereas a more precise, but slower, analysis is needed during compilation. We achieve the variable precision by switching between *intra*-procedural analysis (within a single function definition) and *inter*-procedural analysis (across function definitions). Switching is achieved by constructing two different DFGs, but running the same analyses.

We used MPS-DF in MPS itself and in projects built with MPS to implement several data-flow analyses for C, Java and DSLs.[2] For example, we built analyses to aid developers by marking reads from uninitialized variables and unused assignments in mbeddr C programs, while a commercial project for insurance DSLs uses MPS-DF to empower program understanding.

2. DATA-FLOW ANALYSIS BY EXAMPLE

In this section we illustrate the ingredients of an analysis by developing an example uninitialized read analysis (also called definite assignment analysis) [13]. It marks a variable read as erroneous if the variable can not be guaranteed to have been initialized beforehand. Our target language is mbeddr C and its extensions [19]. mbeddr is a set of languages and an IDE for embedded software development built with MPS.

The code in Figure 2 (A) reads sensor data if the environment is active, otherwise it logs an error message. The call to `calibrateEnv` reads the value of `temp`; however, the read is marked as erroneous, because `temp` is initialized only in the then branch of the preceding `if` statement; `readSensor` dereferences the address of `temp` and assigns a value to it.

2.1 The Data-flow Graph and its Builders

MPS is a projectional language workbench where programs are represented as abstract syntax trees (ASTs). Each AST node is an instance of a *language concept* (meta class). The DFG is an intermediate program representation that represents data-flow information in the program as nodes (for example, `read` and `write` nodes) and encodes the control flow between the nodes as directed edges. The DFG is derived from the AST by data-flow builders. A builder is associated with each language concept and contributes a concept-specific subgraph to the DFG. Builders are expressed using a DSL; using DSLs to express a language aspect is idiomatic in MPS. For example, there are other DSLs for defining a concept's concrete syntax, type checking or transformations.

The DSL for builders shown in Figure 1 relies on three groups of data-flow instructions: intra-procedural, flow-sensitive and inter-procedural.

[2]https://szabta89.github.io/projects/df.html

(instruction)	i	::=	$intra$ | $flow$ | $inter$
(intra-p.)	$intra$	::=	**read** e | **write** $e = e$ | **ret** | **nop** | **code for** e
(flow-s.)	$flow$	::=	**label** id | **jump** p | **ifjump** p **try** e **finally** e **end** |
(inter-p.)	$inter$	::=	**map** $e = e$ | **unmap** e | **entry point** e | **call** e
(position)	p	::=	**before** e | **after** e | **after label** id
(id)	id	::=	identifier
(expression)	e	::=	Java expression

Figure 1: Abstract syntax of the DFG builder DSL.

Intra-procedural instructions encode the basic data-flow of a program. `read` and `write` indicate a read from and a write to a variable, `ret` marks returns from functions and `nop` is an empty operation. `code for` recursively calls the builder of another AST node (e.g. the `then` branch of an `if` statement) and inserts it's subgraph into the DFG at the place of the `code for` instruction.

Each DFG node carries trace information, i.e., it has a pointer to the AST node from which it was created. The trace is automatically set up by the builders. For language concepts that do not affect data flow, builders use the `nop` instruction to support tracing.

Flow-sensitive instructions improve the precision of the DFG by encoding control flow. The unconditional `jump` creates an edge to a DFG node. For example, C's `continue` statement jumps to the start of a loop. The conditional `ifjump` instruction encodes branching in the DFG. For example, C's `if` statement can jump either to the `then` or `else` branch. In the DFG, `label`s are used to mark jump targets. Finally, the `try` instruction encodes the data-flow associated with exception handling.

Inter-procedural instructions help to model the data-flow effects of function calls. An inter-procedural DFG is a hierarchical structure which contains nested DFGs for called functions, where nesting resembles the call stack. `map` passes information from the caller to the callee by assigning an actual argument to a formal parameter. `map` can be imagined as a shortcut for a `read` from the argument and a `write` to the parameter. `unmap` is its counterpart which indicates that a parameter becomes invalid because the called function returns. Builders can contribute `entry points`, which mark nodes as potential targets of calls. Finally, `call` is similar to `code for` with one difference: if the entry point of its called function is present in an ancestor DFG, it creates an edge to the entry point instead of recursively calling the builder of the called function. This is crucial to support modeling recursive function calls, because instead of infinite nesting, only a single edge is inserted.

Example Figure 2 (B) shows the builder of the `FunctionCall` language concept. The code first iterates over the arguments and parameters in parallel and `map`s the arguments to the parameters, thus passing information from the caller to the callee. Next, it uses the `call` instruction to nest the DFG of the called function or to insert a single edge if the `call` is recursive. Finally, the code inserts `unmap` instructions for the parameters. The `mode` property is used for variable precision and it is explained in Section 3.2.

Figure 2 (C) shows the DFG for the code in (A) as it is visualized in MPS-DF. The DFG is hierarchical, because the func-

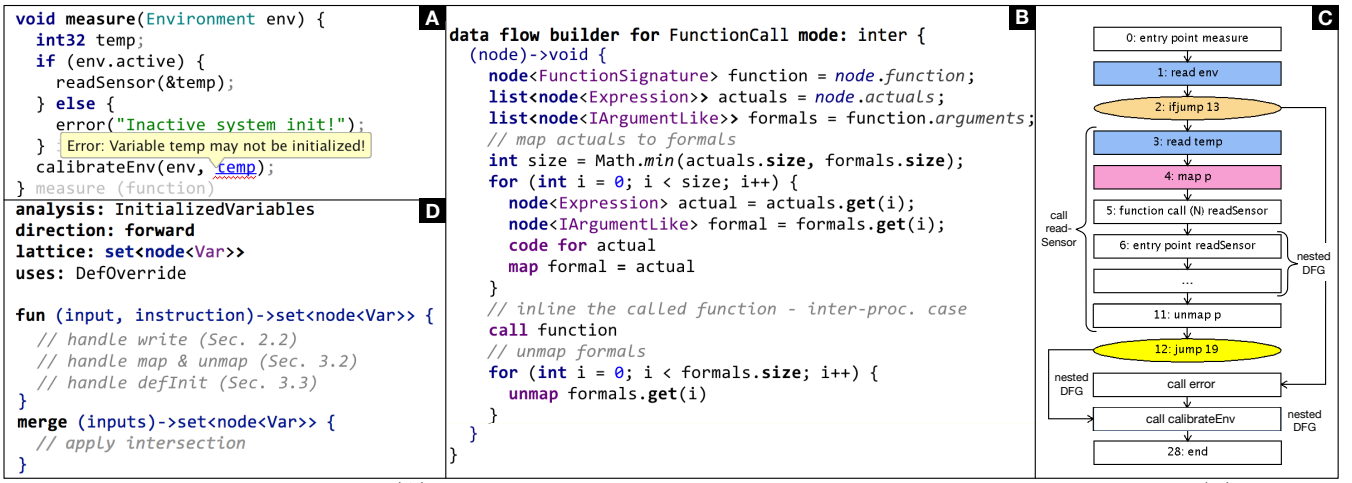

Figure 2: Ingredients of an analysis; (A) a simple C program which we statically analyze for uninitialized reads, (B) data-flow builder of the `FunctionCall` concept, (C) DFG for the code snippet, and (D) the skeleton of the analysis.

tion calls contribute nested DFGs. The call to `readSensor` contributes instructions 3 - 11. For better readability we collapsed the calls to `error` and `calibrateEnv` into a single artificial instruction, but in fact they also contribute a nested DFG similar to `readSensor`. The branching at instruction 2 is contributed by the builder of the `if` statement.

2.2 Analysis Implementation

MPS-DF implements the classical monotone framework for data-flow analysis [11] based on the work list algorithm. In this framework every analysis is characterized by a lattice. A lattice is a partially ordered structure in which every two elements have a unique least upper bound and a unique greatest lower bound. The work list algorithm builds on two analysis-specific functions: `fun` assigns a lattice element to every instruction in the DFG, while `merge` is used to merge lattice elements when control flow edges merge in the DFG. The algorithm uses these two functions and traverses the DFG to assign a lattice element to every node until it reaches a fix point. The fix point computation is required because some control structures (e.g. loops, jumps, recursion) contribute cycles into the DFG, and multiple traversals of these cycles may be required until an assigned lattice element converges. It is the analysis developer's responsibility to define `fun` and `merge` in a way that a fix point can be reached. Certain analysis, e.g. interval analysis [13], may require a widening operator (a form of convergence accelerator) on the lattice to fulfill this requirement.

The `fun` and `merge` functions define how to derive the data-flow knowledge on the DFG of the analyzed program. This knowledge is the analysis result and technically it is a mapping from a DFG node to an element in the analysis-specific lattice. Other MPS components then make use of this result; for example, validation rules create error markers, while a transformation uses it for optimizations.

Approximation Recall that we develop static analyses in MPS-DF. As these analyses do not execute the analyzed programs, they use approximations to derive the analysis result. A *may* analysis is one that derives information that may possibly be true and, thus, computes an upper approximation of the information that would be true during the execution of the analyzed program. In contrast, a *must* analysis computes information that is definitely true and derives a lower approximation. In MPS-DF, the implementation of the `fun` and `merge` functions define whether an analysis is a must or a may analysis. Nevertheless, the must/may property must be coordinated with the client which uses the analysis result. We discuss the relevance of the must/may property on our example analysis next.

Example Similar to builders, the data-flow analysis is also expressed with a DSL. Figure 2 (D) shows the skeleton of the `InitializedVariables` analysis. We define its lattice as `set<node<Var>>` and it encodes the set of definitely initialized variables at a program point. The `forward` direction specifies that the work list algorithm starts the traversal at the first instruction. A `backward` analysis, such as liveness, would start at the last instruction. The `uses` part is used for extensibility and is explained in Section 3.3. The `fun` function builds the aforementioned set of variables for every node in the DFG. When `fun` encounters a `write` instruction in the DFG, it adds the written variable to the set because that variable is now initialized. The `merge` function uses intersection to merge lattice elements, because we develop a must analysis.

The name of the analysis is `InitializedVariables` indicating that it derives information about initialized variables. The analysis is sound, because it claims that a variable is initialized at a DFG node only if it is initialized on all executions paths that lead to that node. The actual error marker in Figure 2 (A) comes from a validation rule of the `Function` concept. The rule uses the must analysis result, iterates over all reads in the function and checks whether the read variable is initialized at the DFG node where the read happens. This is not the case for `temp`, so an error marker is placed. In contrast, a may analysis would use union yielding an unsound result, and, in turn, the dependent validation rule would only mark a read variable as uninitialized if it is not initialized on any of the execution paths that lead to the DFG node representing the read. The benefit of the latter setup would be that it produces less false positives.

3. ARCHITECTURE OF MPS-DF

In this section we present the complete architecture of MPS-DF. We focus on the components that enable variable precision and extensibility.

3.1 Overview

Figure 3 shows the complete architecture of MPS-DF. Com-

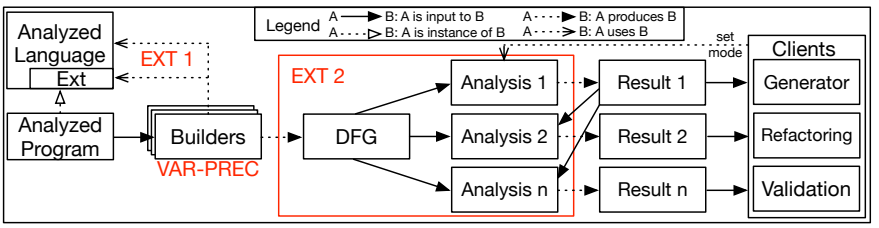

Figure 3: The architecture of the data-flow support in MPS. Annotations mark the components which are responsible for extensibility (`EXT1` and `EXT2`) and for variable precision (`VAR-PREC`).

ponents on the left produce data that serves as input to components on the right. A data-flow analysis always targets a particular language and potentially its language extensions. Given a concrete program of the analyzed language, the data-flow builders construct the DFG. This DFG serves as the input to all data-flow analyses. MPS-DF evaluates the analyses on the DFG and produces the analysis results using the work list algorithm (Section 2.2).

Certain analyses may require the results of other analyses. This happens in our example as well, because the uninitialized read analysis uses the results of a prior points-to analysis to know the possible targets of pointer typed variables. Without this information, it can not know that `readSensor` indirectly initializes `temp`.

There are several examples for clients of an analysis: a code generator can use the result to apply optimizations, analysis results can enable or prevent refactorings and validation rules can create error markers in the IDE on program elements.

3.2 Support for Variable Precision

Variable-precision analyses are inspired by the observation that different use cases of analyses require a different trade-off between precision and performance. While the user edits code in the IDE, she expects the analyses to be fast in order to not break the coding flow; a compromise in precision may be acceptable. However, when the same analysis is used in the compiler, a somewhat longer execution time may be acceptable in order to get better precision through an exhaustive analysis.

Various precision properties (flow-sensitivity, inter-procedurality) can be sacrificed in order to make analyses run faster. For example, the lack of inter-procedurality means that analyses do not know what happens in called functions and thus must approximate the analysis results. The method of approximation is specific to each analysis implementation and must be in coordination with the dependent clients similar to the must/may property (Section 2.2).

Reducing precision has an immediate effect on the size of the DFG, which in turn affects how fast the analyses can be evaluated. In fact, the inter-procedurality yields the largest DFG because it introduces nested DFGs for all called functions. Based on this observation we introduced two modes for analyses: `intra` and `inter` which are used by the builders. We mark the builders with `VAR-PREC` in Figure 3 to indicate that they are the sources of variable precision. When a builder uses the `intra` mode, it contributes to the less precise intra-procedural DFG, and when it uses the `inter` mode, it contributes to the more precise inter-procedural DFG. The two kinds of builders are defined separately for a language concept. Flow-sensitivity can be activated in both modes, because it has only a small effect on the DFG size.

Analyses consume the DFG constructed by the builders. An analysis itself requires only minimal changes to handle the `inter` mode, because only the `map` and `unmap` inter-procedural instructions appear explicitly in the inter-procedural DFG. Nevertheless, builders do not contribute nested DFGs into the DFG in the `intra` mode, thus an analysis does not know what happens in the called functions. In order to mitigate this problem, builders should introduce `nop` instructions for called functions in the `intra` mode. Analyses can then use these instructions to trace back to the originating node for the call in the AST and perform an analysis-specific approximation.

Using the DFG as an intermediate representation has the benefit that we can reuse an intra-procedural analysis and only change its input from an `intra` DFG to an `inter` DFG to derive inter-procedural results. Having both `intra` and `inter` builders and adapting the analysis with a slight change let the clients switch between the two modes and tailor the precision of the analysis to their needs. The exact mode is specified at the client side when the client calls MPS-DF to obtain the results of a data-flow analysis.

Example The `FuctionCall` builder in Figure 2 (B) is defined for the `inter` mode, which means that it contributes to the inter-procedural DFG of the program. For the `intra` mode we define a builder that does not use any inter-procedural builder instruction, but contributes a `nop` instruction to make tracing possible.

The `InitializedVariables` (Figure 2 (D)) analysis requires only a slight change in the `fun` function to support the `inter` mode. Whenever it sees a `map`, it adds the written variable to the set of initialized ones. If it encounters an `unmap`, it simply removes the given parameter from the set to forget about it. In the `intra` mode, in accordance to the must property, the `InitializedVariables` analysis handles a `nop` instruction which traces back to a function call by assuming that no variable is initialized in the called function. This preserves soundness, because we do not add any variable to the set of initialized ones for which we can not guarantee the initialized property.

3.3 Support for Extensibility

MPS provides extensive support for composing languages [18]. In order to not limit this support, data-flow analyses must be extensible as well. We address this requirement with two flavours of extensibility.

Builders enable one form of extensibility (`EXT1` in Figure 3): the DFG becomes extensible in the face of language extensions. When MPS-DF constructs the DFG of a program, it executes all builders of the involved language concepts. As long as a language extension contains builders for their extension concepts, these are automatically taken into account, resulting in a DFG for the program written in the composed language. Typically, an existing analysis will also work for programs of an extended language. This is because the analyses are evaluated on DFGs only and it does not matter how that DFG was built.

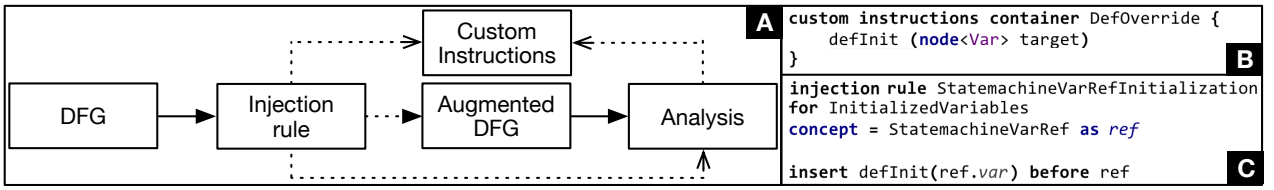

Figure 4: Components supporting extensibility of analyses in the face of language extensions. Subfigure (A) unfolds the `EXT2` annotation in Figure 3 and uses the same legend as Figure 3. Subfigure (B) shows our example custom instruction and (C) shows the example injection rule for the `StatemachineVarRef` concept.

There is another flavour of extensibility: the analyses may require extensibility in the face of language extensions. In other words, sometimes it is necessary to augment the DFG with custom nodes to support a particular analysis by encoding domain-specific knowledge. This form of extensibility is achieved through the combination of two components: *custom instruction* and *injection rule*. Figure 4 (A) shows their connection. First, an analysis developer defines a set of custom data-flow instructions for a particular analysis. Then, an analysis developer defines injection rules for the analysis. A rule is similar to a builder as it also contributes nodes and edges to the DFG for a language concept, but it can only inject the custom instructions of its analysis. If an analysis has injection rules, then its input is not the common DFG anymore, which is shared by all rule-less analyses, but the one which is augmented by its rules.

Example mbeddr C comes with a language extension for inline definitions of state machines. It is possible to create global variables of type state machine and, like any other global variable, they can be referenced from C code. However, the code generator of the state machine extension makes sure that a state machine variable is *always* initialized. To avoid false positive uninitialized read errors, we use custom instructions in the DFG to override the analysis result.

We define one custom instruction, `defInit` (Figure 4 (B)), and organize it into a container `DefOverride`. The instruction points to a variable, and its presence in the DFG means that the pointed-to variable is definitely initialized. There are two components in the architecture which use `defInit`. A rule which augments the DFG and the analysis which handles the custom instruction in the augmented DFG. Figure 4 (C) shows an injection rule for the `StatemachineVarRef` concept and for the `InitializedVariables` analysis. The rule injects a `defInit` instruction for the referenced variable of a `StatemachineVarRef`. The `before ref` means that the `defInit` is inserted just before the subgraph contributed by the builder of the reference.

It is the responsibility of the analysis to handle the custom instruction and indirectly let the rule override its result. Figure 2 (D) shows that the analysis uses the `DefOverride` container to access the contained custom instructions. It handles a `defInit` instruction in the `fun` function similar to a `write` as it adds the pointed-to variable to the set of definitely initialized variables. As there is a `defInit` before every read from a state machine variable, the validation rule will not mark any of these reads as erroneous.

4. DISCUSSION

Validation MPS-DF is in heavy use in both MPS itself and in the mbeddr IDE to develop data-flow analyses. For example we developed points-to, uninitialized read, liveness analyses for mbeddr C and null analysis for Java. The analyses implementations are available in our online material.

Additionally, language engineers (and not the developers of MPS-DF) at the company itemis have used MPS-DF to implement data-flow analyses in several customer projects in the domains of embedded systems, insurance, and high performance computing. Based on their experience and feedback, the data-flow analyses indeed evolve together with the developed languages. Adding new language extensions requires data-flow related extension as well. If the new language constructs do not provide builders then they do not contribute to the DFG, which will be immediately visible in the form of false positive analysis results on the extended program. On the other hand, due to the support for extensibility, the mitigation usually ends with implementing new builders and/or rules.

Performance We evaluated the performance[3] of MPS-DF on the Toyota ITC benchmark,[4] a collection of C code snippets with intentional bugs to test the precision of static analysis tools. The code base comprises about 15,000 lines of C code, which we imported into mbeddr. We ran the uninitialized read analysis together with a points-to analysis on the complete code base in both `intra` and `inter` modes. The `intra` analysis requires 8.9 seconds on the complete code base and the `inter` analysis needs 96.6 seconds, which is a 11x slowdown. This shows that a more precise `inter` analysis requires considerably more time. The source of this slowdown is the inlining of nested DFGs in the `inter` mode, which could be mitigated by the application of summaries as in Soot [16].

Portability to other IDEs We designed a generic architecture for data-flow analysis that can be used in other language workbenches as well. However, the explicit construction of the DFG may have an effect on the performance in other workbenches. MPS-DF builds the DFG from the AST of the analyzed program, thus the AST must be available first. As data-flow analyses may run frequently (even after every code change), an up-to-date AST must be always available for the analyses. This kind of incremental maintenance is not an issue in projectional workbenches (e.g. MPS or Eco [4]) where the AST is always available and users directly modify it with tree transformations. However, parser based systems may be able to cope with this challenge only with an incremental parser in the background (cf. the survey by Ramalingam and Reps [14] for example incremental parsers). Nevertheless, the requirement for the AST is not specific to MPS-DF analyses, because program analyses are usually carried out on the AST.

[3]We ran the measurements on a 64-bit OSX 10.10.3 machine with an Intel Core i7 2.5 GHz processor and 16 GB of RAM using Java 1.8.0_65.
[4]https://github.com/regehr/itc-benchmarks

5. RELATED WORK

There are several tools which rely on relations and relational algebra to carry out program analyses; DeFacto [1], CrocoPat [2] and Grok [10]. Similar to MPS-DF, these tools also decouple the creation of an intermediate program representation and the analysis. The program representation is tuples of relations and these tools all come with a language for the definition of relational operators which derive the analysis result. All these tools are independent of the analyzed language similar to MPS-DF. Nevertheless, they were not designed with extensibility in mind and target only one particular language without considering language extensions.

DCFlow [9] is a DSL and Rascal library for constructing control flow graphs of programs. The DSL is similar to the flow-sensitive part of our builder DSL. However, in DCFlow only the control flow graph is extracted as an intermediate representation and not the primitive data-flow instructions (e.g. read, write), which prevents extensibility of analyses. Handling a new kind of language concept requires invasive changes in the analysis implementation. In contrast, MPS-DF only requires a new builder implementation and no modifications to the analyses.

Bodden et al. extend the Soot [16] framework to support inter-procedural data-flow analyses [3]. Compared to our solution of nesting the DFG of a called function, they use summaries: instead of reanalyzing a function for all call sites, a summary is computed once to capture the effects of a function and it is applied and reused for all callers. This could lead to great performance improvements for large programs. From the user perspective the tool is programmable through Java APIs and it can analyze only Java programs. In contrast, MPS-DF supports variable-precision data-flow analyses and it is independent of the analyzed language.

6. CONCLUSIONS

We presented MPS-DF which is the data-flow support in MPS. It defines a builder DSL for the construction of variable-precision DFGs and an analysis DSL for the definition of data-flow analyses. Additionally, MPS-DF analyses are extensible in the face of language extensions of the analyzed language.

We found MPS-DF useful in several open-source and commercial projects centered around DSLs for embedded systems, insurance and high performance computing. MPS-DF constitutes an integral part of the MPS language workbench and it is available open-source.

7. REFERENCES

[1] H. J. Basten and P. Klint. Defacto: Language-parametric fact extraction from source code. In *Software Language Engineering*. Springer-Verlag, 2009.

[2] D. Beyer. Relational programming with crocopat. In *Proceedings of the International Conference on Software Engineering*. ACM, 2006.

[3] E. Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the International Workshop on State of the Art in Java Program Analysis*. ACM, 2012.

[4] L. Diekmann and L. Tratt. *Eco: A Language Composition Editor*. Springer International Publishing, 2014.

[5] J. Dietrich, N. Hollingum, and B. Scholz. Giga-scale Exhaustive Points-to Analysis for Java in Under a Minute. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2015.

[6] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM, 1994.

[7] S. Erdweg, P. G. Giarrusso, and T. Rendel. Language Composition Untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*. ACM, 2012.

[8] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning. Evaluating and comparing language workbenches. *Comput. Lang. Syst. Struct.*, 2015.

[9] M. Hills. Streamlining Control Flow Graph Construction with DCFlow. In *Software Language Engineering*. Springer International Publishing, 2014.

[10] R. C. Holt. Binary relational algebra applied to software architecture. *Computer Systems Research Institute*, 1996.

[11] G. A. Kildall. A Unified Approach to Global Program Optimization. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM, 1973.

[12] J. Kärnä, J.-P. Tolvanen, and S. Kelly. Evaluating the use of domain-specific modeling in practice. In *Workshop on Domain-Specific Modeling (DSM)*. 2009.

[13] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

[14] G. Ramalingam and T. Reps. A Categorized Bibliography on Incremental Computation. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM, 1993.

[15] M. Sridharan and R. Bodík. Refinement-based Context-sensitive Points-to Analysis for Java. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM, 2006.

[16] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press, 1999.

[17] E. Visser. Webdsl: A case study in domain-specific language engineering. In *Generative and Transformational Techniques in Software Engineering II*. Springer-Verlag, 2008.

[18] M. Voelter. *Language and IDE Modularization and Composition with MPS*. Springer Berlin Heidelberg, 2013.

[19] M. Voelter, D. Ratiu, B. Kolb, and B. Schaetz. mbeddr: Instantiating a language workbench in the embedded software domain. *Automated Software Engineering*, 2013.