XRT– Exploring Runtime for .NET
Architecture and Applications

Wolfgang Grieskamp Nikolai Tillmann Wolfram Schulte

Microsoft Research, Redmond, USA [1]

Abstract

XRT– E**x**ploring **R**un**t**ime – is an exploration framework for programs represented in Microsoft's common intermediate language (CIL). Processing .NET managed assemblies, it provides means for analyzing, rewriting, and executing the rewritten program. Whereas XRT's representation of state allows for arbitrary exploration strategies, it is particularly optimized for transactional exploration, where a transaction may consist of many instruction steps. XRT supports extensions, and one such extension is a module for symbolic exploration which captures the complete domain of safe CIL. Current applications of XRT are in the area of testing, namely parameterized unit testing and state-space exploration for model-based testing. This paper gives an overview of the architecture of XRT and outlines the applications.

Key words: CLR model-checking, extensible exploration framework, mixed concrete/symbolic state

1 Introduction

Recently, there has been increasing interest in the application of model checking to software. Software model checking technology allows detecting difficult to find bugs like data races, and verifying that models and/or implementations satisfy crucial temporal properties.

The effectiveness of the model checker for finding bugs typically depends on the application domain. For instance, SPIN [13] works very well for checking models of protocols, the SLAM [4] and BLAST projects are successful since they focus on device drivers, VeriSoft [11] is successful in checking certain software which can't be mapped into a model-checker's input language by re-execution, while Zing [3] is successful on checking refinements of concurrent object-oriented abstractions. *Specialization* for different application domains seems to be a key for successful application of software model-checking.

[1] Email: wrwg@microsoft.com, nikolait@microsoft.com, schulte@microsoft.com

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

However, most of the existing model checkers are monolithic, i.e. state representation and exploration is highly intertwined and the input language is fixed. The Bogor project [17] and the Java Pathfinder project [1] are addressing this problem by providing an open framework which can be specialized for different kinds of application domains.

XRT is a new state exploration framework that follows similar goals as Bogor and JPF, using the approach of execution on the virtual machine level as pioneered by JPF. It supports the full safe (verifiable) intermediate language of the Common Language Runtime (CLR), CIL, and provides extension points on various levels, including the instruction set, the state representation, and the exploration strategy. It has been developed from the beginning together with one particular extension in mind, namely the unrestricted support of mixed *concrete/symbolic state and exploration*, where logical variables can range over all values appearing in a CIL program, including objects and arrays.

XRT's development was motivated mainly by applications in the testing realm. Together with the theorem prover ZAP developed at MSR [21], it provides the basis for MUTT, a project around techniques and tools for unit testing, with first outcomes described in [22]. It also serves as the core for the next version of the model-based testing tool Spec Explorer [5], as described in [12].

This paper gives a high-level overview of the architecture of XRT for a technically interested and informed audience. We first outline the core architecture, then describe the extension for symbolic exploration, and finally sketch the current applications. Augmenting technical material is found in appendices.

2 Core Architecture

The core of XRT consists of three major subsystems which provide the *program model*, the *state representation*, and the *execution environment*. These subsystems are based on a strict, generic component model, which is also expected to be used by XRT applications: public functionality is described and exposed by service interfaces, and components using this functionality query implementers via their service interface types, ensuring component substitutability.

2.1 Program Model

A program in XRT consists of assemblies. An assembly can be loaded explicitly, or as a side effect during program execution using standard .NET assembly loading rules. The elements of assemblies are loaded and instrumented lazily. A method's byte code, for example, will not be loaded and rewritten before the method is executed the first time. This way, XRT can run on a large code base (like e.g. *mscorlib.dll*, the core assembly of .NET) which is only processed as far as the dynamic control flow requires.

Meta Data The program model represents meta data in a conventional way: for each of the elements of an assembly – types, methods, fields, locals – an according type exists. .NET custom attributes on each of the entities, including an assem-

bly itself, are available, which is often useful for instrumentation of the processed program.

Code Representation The basic entity of executable code is a method body, which is represented as a *control flow graph*, with nodes being basic blocks. Each basic block ends either in a (sequence of) branch instructions (pointing to other blocks), in a return instruction, or in an unwind instruction, and has as a special *exception exit* which points to the block being executed if an exception is raised. If an exception is not handled in a particular block context the exception exit will point to a block which just contains an unwind instruction.

Instructions are represented by a language we call XIL, which is an abstraction of CIL. XIL is similar to three-address code and operates on local variables, which stem from the local variables and parameters in the original program, as well as from temporaries which have been introduced for evaluation stack locations.

XIL deals with all concepts of safe, verifiable CIL, including addresses for methods, locals, fields, and array elements. The instruction set is documented in Appendix A. There is one special XIL instruction which should be mentioned here since it provides the major extension point: `CALLPRIMITIVE` $p(\bar{l})$, where p is a delegate pointing to some meta-level code, and $\bar{l}$ is a sequence of local variables. To interpret this instruction, the execution engine calls the delegate as discussed later.

Flavors and Instruction Rewriters A method can have many different *flavors*. Flavors allow several code versions – for example, a series of refinements – of the same method in one XRT exploration session. A flavor defines a pipeline of *instruction rewriters* which are run on the initial XIL version of the method's code the first time the flavor of a given method is requested. Each instruction rewriter takes the control flow graph of a method body and maps into another one. In the simplest case, it can just perform some instruction-local substitutions; more sophisticated rewriters may produce a completely new control graph. The infrastructure supports rewriters in various ways. For example, a data flow analysis framework is provided on top of the control flow graph representation which allows analyzing the method body prior to rewriting.

Method and Type Substitution An alternative to instruction rewriting is method or type substitution. For method substitution, one can provide a meta-level delegate which is called when the method is invoked instead of interpreting the method's instructions. This is used for example to substitute native method calls on framework types like `String`. Commonly, the method substitution will invoke the substituted method's native implementation using reflection. This is possible for methods without side-effects which operate on data representations which can be converted to native form (like e.g. numbers and strings).

Type substitution replaces an entire type, including its heap representation, by another type which has a compatible signature. This is transparent in the program model, i.e. when the substituted type or one of its methods is queried, the substitu-

tion will be delivered, and a substituted class can be base class of a non-substituted type.

Type substitution can be done either in a declarative way by means of a custom attribute which states that one type in the set of loaded types is replaced by another, signature-compatible one. This approach is used when the replacement can be realized in standard C#/.NET code which is subject of regular execution under XRT, for example to deal with native type implementations. Type substitution can also be done in a programmatic way by calling into XRT's program component, registering new type representations and individual method call handlers which have full access to the internals of XRT. Programmatic substitution is often used to replace types which have a special meaning for exploration, like .NET threads. Appendix B contains an example of programmatic type substitution for the .NET framework type `System.Random`.

2.2 State Representation

XRT's state encodes a full snapshot of the program's state, including static data, heap, and threads with call stacks. XRT distinguishes two state representations. An *active* state is a mutable version of the state which allows full reading and writing access. A *compressed* state (also called collapsed state in the literature) is an immutable version which allows for fast hashing and comparison. Active states can be obtained by uncompressing compressed states, and vice versa.

Compression In the current implementation, the compressed state is realized as an internalized (hash-consed) vector of components. Uncompressing a compressed state is cheap, since initially the active state is just a wrapper around the compressed state from which it is derived. Only when components are accessed in the active state which live in the compressed state, they are fetched from the compressed state, uncompressed, and cached in the active state. For example, the active state may process a sequence of method calls on top of the call stack where lower parts of the call stack stay untouched in the compressed state.

When an active state is compressed, a new compressed state will be created based upon the delta of the changes performed in the active state compared to the compressed state from which it was derived. Compression is similar to copying GC algorithms in that it walks over the reachable object graph of the active state and relocates reachable objects into the new compressed state. Only at this time, and only for the active, reachable values, structure sharing needs to be computed; for unchanged parts, it is taken over from the old compressed state.

Garbage Collection and State Symmetries The current state implementation uses a reference counting mechanism to detect dead objects in a compressed state, which is approximative because of the potential presence of cycles in object graphs. The reference count needs to be only maintained during compression. When the hash-code is computed, or the heaps of a given object type of two compressed states are compared, reference count information is used to skip dead entries. Provided the dead objects are at the "end" of the heap (are younger than other objects) this

heuristic detects some object graph symmetries in an inexpensive way.

Global garbage collection on *all* living compressed states is used to prevent the global background, which contains the internalization tables, to grow indefinitely. It also detects symmetries regarding those state components which can be deterministically ordered. However, it must be only performed occasionally, so that its cost amortizes over an exploration session. More work needs to be done for XRT w.r.t. detecting symmetries incrementally and using heuristics for component ordering, most likely adapting previous work done in this area (e.g. [14,20,18]).

State Extensions The core state provides a way to plug-in *state extensions*. To that end, it distinguishes for the encoding of every value whether its representation is in the core or in the extension. All interpretations on extended values (which can be of value or reference type) are forwarded to the state extension. Like the state, an extension has an active and a compressed representation, and the compression algorithm of the core calls into the extension and is called back as required.

The interfaces to be implemented by a state extension are simpler than those of the core state, which justifies that custom states are not just implemented by replacing the entire core state implementation. In particular, the state extension does not need to deal with threads, call stacks, and addresses of locations, but only with the representation of values and objects. The major application of the state extension is currently in adding symbolic state to XRT, as discussed in Section 3.

2.3 Execution Environment

The basic facility for executing code is the executor component. Among others, it provides a method which takes an active state and iterates instruction stepping of the active thread (including calls and returns) until a suspension point is hit.

Suspensions can be triggered by the delegate of a `CALLPRIMITIVE` $p(\bar{l})$ instruction, or a method substitution's delegate; we call these delegates *primitive call handlers*. When called by the executor, such a handler can return an object which represents a *suspension*; in this case the executor will stop iterating and pass the suspension object to its caller, usually an exploration algorithm.

Suspensions and Exploration A suspension can act like a choice point in a state exploration graph. It can capture a compressed state and an enumeration of outgoing *transactions*. (We use the notion of transaction instead of transition or step in order to emphasize that a transaction may consist of many logical state transitions/instruction steps.) The transactions of a suspension are not yet computed when enumerated. Instead, the suspension provides a computation method for its transactions which will uncompress the state contained in the suspension and call the executor to continue execution in the given path until the next suspension is hit.

Suspensions are given by interface types, and many different implementations may exist in one generic exploration algorithm. The XRT framework provides standard suspensions representing the activation of a top-level entry point method (the initial node of an exploration graph), thread scheduling suspension, and others, and it facilitates the creation of custom suspensions. In Appendix B we show a

code sample which, among other things, illustrates the implementation of a custom suspension.

Re-Execution Typically, if a suspension represents a branch point for exploration, the instruction which causes the suspension should be interpreted again in each branch under different conditions. To that end, the compressed state provides a flag on a per-thread basis to let a primitive call handler trigger the re-execution of its instruction. The active state provides information about the transaction and the associated suspension in which the current instruction is executed. When the re-execution flag is set, it is ensured that the transaction seen by the primitive call handler belongs to the suspension it has created when it was executed the first time. This way, internal state can be communicated from the initial execution to the re-execution point. The usage of re-execution is illustrated by the sample in Appendix B.

3 Symbolic State Extension

One extension of XRT's core state realizes *symbolic state*. This allows to represent values of arbitrary .NET types symbolically. We next outline the design of this extension. We call XRT plus its symbolic state extension $\text{XRT}_{\mathcal{S}}$.

Terms Symbolic values are represented by internalized (hash-consed) *terms*. There are the usual terms for ground values, logical variables, object references, object and array state, unary and binary operations of CIL, "struct" values (free constructors), and so on. Here we only discuss three special forms of terms: type, object state and domain terms. The whole term language is documented in Appendix C.

$\text{XRT}_{\mathcal{S}}$' term language includes terms for *types* and *type constraints*. Each term has an associated base type. If the base type is a reference type, the runtime type of the object it represents might be more specific, and can be symbolic as well.

Object state is represented by so-called *field maps*, array state by so-called *element maps*. Both representations are adopted from the theorem prover community [7]. A field map is logically a function mapping object terms to terms representing the field assignment for the object; similar, an element map is a function from object terms and index terms into terms for element assignments. The field and element maps are syntactically defined by a series of updates on initial map values. See Appendix C for the details.

A *domain term* represents a set of values. Constructors for domain terms are the empty set, the singleton set, the range set (for numbers), and set union. We use domain terms in $\text{XRT}_{\mathcal{S}}$ to represent membership constraints of terms. For example, if a new logical variable is created whose base type is a reference type, if not otherwise specified, this variable ranges over the object references available for that base type in the state where the variable is created; and this fact is expressed by a domain constraint.

Terms are always constructed and analyzed using a so-called *term manager* which is associated with a solver. The default implementation of the term manager performs internalization and normalization (including simplification) of terms. The

solver can override this implementation by adding further normalizations as required.

Solvers and Assumption Sets A solver is a component in the XRT_S framework whose API exposes the construction of *assumption sets*, which represent an abstract set of constraints. Like states, assumptions come in compressed and active form. Operations on assumptions include term simplification, satisfiability and subsumption check, domain query, and assumption split. The result of each of these queries can be inconclusive.

Simplification takes a term and rewrites it to a simpler term, which may become ground this way. Satisfiablity checks whether a given assumption set has a model. Subsumption checks that one assumption set is subsumed by another (i.e. the set of models is a subset of the set of models of the other).

A query for a term's domain is handled as follows. If an explicit domain constraint for the term is in the assumption set, it determines the term's domain. Otherwise, a *derived* domain of the term can be computed for every compound term, provided the subterms have a domain. Consider, for example, a term which represents the addition of two logical variables which have range constraints. The derived domain of the addition term can be computed and will range from the sum of the individual ranges' starts to the sum of their ends. Derived domains are approximations of the actual domains, which may be further restricted by other constraint. Term domains are used for assumption set splitting.

Splitting is performed relative to a term and results in an enumeration of new assumption sets in which the given term has a more specialized domain. In more detail: if the domain *d* is a union, then the resulting assumption set will represent the left and right operand of the union; if *d* is a range, the range will be split in the middle. For example, let *t* be a Boolean term with domain $t \in \{0\} \cup \{1\}$ where 0 and 1 represent **false** and **true**, respectively. Splitting over *t* will produce two new assumption sets, one which contains $t \in \{0\}$ and one which contains $t \in \{1\}$. For each case the solver now has complete knowledge, and can, for example, simplify terms further based on the split assumption set.

Solvers can be stacked, where one solver makes use of an underlying solver. XRT_S comes with a default solver implementation which supports quick decision procedures on membership constraints (and thereby equalities and unification) and supports domains and splitting, and which can leverage an underlying solver for satisfiablity and subsumption checks. The underlying solvers used in this configuration are currently either Simplify [7] or ZAP [21]. If no underlying solver is present, the default solver will use for satisifiablity checks finite-domain solving techniques [16], if applicable, and, as the last resort, global search.

Symbolic State and State Extension A symbolic state consists of an assumption set, and a mapping from fields and array types to their current field and element maps. There is one field map for each instance field, and one element map for each value type, but only one element map for all arrays with reference types as elements to allow for covariance. Initially, field and element maps can either be empty

(*closed world*), thereby restricting field and element accesses via a logical variable of a reference type to range only over subsequently created objects, or they can be represented by logical variables (*open world*), thereby creating an unbounded symbolic universe accessible with logical variables of reference types.

A symbolic state can live independently of a XRT core state, so that it can be used in contexts where no core state is available. The *symbolic state connector* wires a core state to a symbolic state using XRT's state extension API. The major conceptual responsibility of the connector is to deal with compression, and the synchronization of the object world of the core state with the symbolic state. For the last point, the core state and the connector work together to realize *migration* of objects from the core state into the symbolic state, preserving the object's identity in the core state. Migration is necessary, for example, when a field update is perfomed on a logical variable of a reference type. This update could potentially address any of the objects in the domain of that variable, and we therefore need to migrate all those objects to the symbolic state. The opposite, migration from the symbolic to the core state, is not always possible. Only if symbolic objects are ground, they can be brought back into the concrete state by triggering garbage collection.

Symbolic Exploration Symbolic exploration is realized by an instruction rewriter (cf. Section 2.1) and a custom suspension added to the execution environment (cf. Section 2.3). The rewriter on the one hand substitutes certain instructions by their symbolic counterparts, for instance unary and binary operators, the is-instance check, and the lookup of virtual method addresses; on the other hand it introduces *checkpoints* before instructions with multiple control-flow exits.

XRT$_{\mathcal{S}}$' symbolic exploration supports a variety of checkpoints. For example, for a conditional branch, a checkpoint is created which tests whether the condition evaluates to true or false in the current assumption set; if the check is inconclusive, symbolic execution creates a *symbolic suspension*. The resulting transactions represent a split of the assumption set over the branch condition. Other instances of checkpoints include type tests, non-null tests, and so on. The complete set of checkpoints is documented in Appendix D.

Note that satisfiability checks are *only* performed at checkpoints; in fact this is the only place where assumptions can be strengthened during symbolic exploration.

4 Current Applications

We sketch two promising applications of XRT; both of which are in the realm of testing, and motivated the construction of the framework.

Exploration for Model-Based Testing In previous years, we have developed a tool for model-based conformance testing, called Spec Explorer [5], which is now used on a daily basis by Microsoft product groups mainly to test parts of Windows, and web service infrastructure. This tool uses state space exploration on the model to test an implementation for conformance, either offline (by producing a test-suite from the model exploration) or online (by folding the exploration process with the conformance check). To realize model exploration, Spec Explorer uses special

compilation techniques for its modeling language, Spec#.

The next generation of Spec Explorer which is currently being developed is based on XRT. With XRT, we address our customers' essential requests for improvements, among which are independence of a particular modeling language (customers want to use C# or VB to write models), and stronger means for *scenario control* and *model composition*. Regarding the last point we have developed a framework of so-called *action machines* [12] which are implemented on top of $\text{XRT}_{\mathcal{S}}$ and emphasize the composition aspect. Action machines encode state transition systems, ranging from abstract state machines, statecharts, scenario/use case machines, to implementations. Action machines can be composed using combinators for product, conformance (alternating refinement [2]), and action refinement.

The implementation of action machines in XRT benefits from instruction rewriting and the use of symbolic state as the glue for composition. Each action machine runs its own internal state explorer, avoiding the explosion of the interleaving of internal steps. In a composition, action machines synchronize over simultaneous steps, where a step is labeled by an action given as a term and a set of assumptions. The product machine of two action machines performs a step if action terms of the two machines unify and their assumptions sets under the yielding substitution are satisfiable.

Instruction rewriting is used, for example, in the implementation of scenario machines. A scenario is represented by a C# or VB program which invokes the methods of the "actors" of the system like regular method calls, intertwined with assertions and assumptions about the passed parameters. For exploring the scenario independently of an actual implementation of the actors, the actor method calls are abstracted by an instruction rewriter and replaced by suspensions which produce according steps of the action machine. For details, see [12].

Parameterized Unit Testing Parameterized unit tests [22] extend the current industry practice of using closed unit tests defined as parameterless methods. Parameterized unit tests separate two concerns: 1) They specify the external behavior of the involved methods for all possible test arguments. 2) Test cases can be re-obtained as traditional closed unit tests by instantiating the parameterized unit tests. In addition, parameterized unit tests can be interpreted as symbolic summaries of the involved methods' behavior. These summaries serve as custom rewrite rules, which allows symbolic execution to scale for arbitrary abstraction levels.

In our prototypical realization, symbolic exploration under $\text{XRT}_{\mathcal{S}}$ is used to calculate the parameter instantiations which are necessary to cover all paths – within certain bounds – of the tested implementation. Moreover, symbolic summaries are created from parameterized unit tests by exploration under $\text{XRT}_{\mathcal{S}}$ as well, using a special instruction rewriter to abstract the code appropriately. A call to a summarized method is interpreted as an update on a summarized heap, and the call's normal or exceptional result is encoded as a term referring to the summarized heap and the method's arguments at the time of the call. $\text{XRT}_{\mathcal{S}}$'s solver is able to leverage the summaries as rewrite rules on these terms. This allows to use the parameterized

unit tests of one underlying module as axioms in the parameterized unit tests of another module. For details, see [22].

5 Discussion

Related Work Model checkers like Spin [13] or Zing [3] use a compilation approach instead of an interpretation approach as found in JPF [1], Bogor [19] and XRT, which results in a rather different architecture which is hard to relate. In general, we believe both approaches, compilation and interpretation, have their advantages and disadvantages. One advantage of the interpretation approach is its lower complexity, in particular regarding extensions. On the other hand, compilation might ultimately reach better performance; however, we believe in order to really exploit this potential, one must compile down to a language like C or C++ which provides full access to the machine model.

JPF [1] has pioneered the approach of exploration on the level of a standard virtual machine. As JPF targets the JVM and XRT the CLR, the language treated by XRT is richer, including user-definable value types and addresses for reference parameters and for methods.

Bogor [19] is closely related to XRT regarding the ambition to provide an extensible framework for state exploration from the beginning, whereas JPF was originally designed as an explicit state model checker, and only in recent years developed into a general exploration framework. Bogor and XRT share first design principles like a strictly componentized architecture where the central modules of the system can be exchanged by customized implementations. However, if it comes to the details of the extension points, the systems differ. Bogor uses its own input language, BIR, which provides a mechanism for defining type extensions whose signatures are specified in BIR and whose implementation is given in some Java module. This extension mechanism is comparable to XRT's programmatic type substitutions. JPF provides a way to replace one Java type by another Java type, which is comparable to XRT's declarative type substitutions. However, both Bogor and JPF do not seem to support a systematic approach to code rewriting, as given by XRT's code flavors and instruction rewriter pipelines. We experience that instruction rewriting is a major extension facility for all of our current applications.

Another difference between Bogor as well as JPF and XRT's extension mechanisms seems to be the ability of XRT to compute the length of transactions (atomic computation steps) dynamically. In XRT, a primitive call handler can decide on base of the current program state whether a suspension/choice point will be created. For example, in symbolic exploration a choice point for a branch will only be created if the branch condition is not determined, i.e. the solver cannot reduce the condition to either true or false in the current context. In Bogor, the length of transactions of a thread is fixed by BIR. The ability of dynamically computing transactions can be considered significant in the context of dynamic reduction techniques.

JPF's and Bogor's implementation of state use state collapsing, as does XRT. Collapsing became popular with its implementation for Spin. Currently, XRT's

implementation of collapsing is not as advanced as JPF's and Bogor's as it comes to detecting state symmetries as described e.g. in [20]. Future work on XRT will need to deal with this.

Orthogonal to detecting state symmetries is partial-order reduction [6], which has been investigated in the context of object-oriented concurrent software with threads and locks e.g. in [10,8]. We have a prototype for full dynamic partial-order reduction in XRT which extends the work in [9] to stateful exploration and arbitrary search strategies; however, this prototype requires further investigation before going to publication.

As it comes to hybrid concrete/symbolic exploration, there is an extension to JPF which is similar to XRT$_\mathcal{S}$ [15], and one for Zing is also in preparation. However, these extension do not tackle full symbolic representation of objects; rather, objects are enumerated, and only value types can be symbolic. The designers of Bogor argue that extensions of Bogor with symbolic state are desirable, but *may* require refactoring of their architecture [17]. To the best of our knowledge, XRT is the first realization of full mixed concrete/symbolic exploration, including the handling of symbolic objects and arrays, and allowing exploration in the closed as well as the open object world.

Performance Benchmarking is a black art, at least. Nevertheless, more empirical data is needed to systematically evaluate the performance of XRT and XRT$_\mathcal{S}$. At this point, the current applications suggest that XRT's performance is very reasonable at least for the given domains. For example, parameterized unit testing is able to generate the parameters which provide path coverage for interesting scenarios of a relative complex and nonorthogonal implementations like hashtables in under 30 seconds. As we will get more applications running on top of XRT, we will have more opportunities to evaluate the efficiency of XRT under real loads.

Future Work XRT needs further improvements. One important topic is to add better means for detecting state symmetries, another is to consolidate the dynamic partial-order reduction technique prototyped in XRT. A further area of interest is *goal-oriented* and *heuristic* search, which is of importance for testing as well as model-checking; to this end we are looking at techniques based on program slicing and try to extend them using our symbolic computation framework. Finally, we haven't yet instantiated XRT for model-checking applications, but plan to do so at some future point. Currently our major focus is on pushing the applications for model-based testing and unit testing.

References

[1] *Java PathFinder Home Page*, http://javapathfinder.sourceforge.net/.

[2] Alur, R., T. A. Henzinger, O. Kupferman and M. Vardi, *Alternating refinement relations*, in: *Proceedings of the Ninth International Conference on Concurrency Theory (CONCUR'98)*, LNCS **1466**, 1998, pp. 163–178.

[3] Andrews, T., S. Qadeer, S. K. Rajamani, J. Rehof and Y. Xie, *Zing: Exploiting program*

structure for model checking concurrent software, in: *CONCUR 2004*, 2004.

[4] Ball, T. and S. K. Rajamani, *The SLAM project: Debugging system software via static analysis*, in: *POPL 2002*, 2002.

[5] Campbell, C., W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann and M. Veanes, *Model-based testing of object-oriented reactive systems with Spec Explorer*, Technical Report MSR-TR-2005-59, Microsoft Research (2005), submitted.

[6] Clarke, E. M., O. Grumberg and D. Peled, "Model Checking," MIT Press, 1999 .

[7] Detlefs, D., G. Nelson and J. Saxe, *Simplify: A theorem prover for program checking* (2003). URL citeseer.ist.psu.edu/detlefs03simplify.html

[8] Dwyer, M. B., J. Hatcliff, Robby and V. P. Ranganath, *Exploiting object escape and locking information in partial-order reduction for concurrent object-oriented programs*, Formal Methods in System Design **25** (2004).

[9] Flanagan, C. and P. Godefroid, *Dynamic partial-order reduction for model checking software*, in: *POPL'05*, 2005.

[10] Flanagan, C. and S. Qadeer, *Transactions in software model-checking*, Electronic Notes in Theoretical Computer Science (2003).

[11] Godefroid, P., *Software model checking: The VeriSoft approach* .

[12] Grieskamp, W. and N. Tillmann, *Action machines – towards a framework for model composition, exploration and conformance testing based on symbolic computation*, Technical Report MSR-TR-2005-60, Microsoft Research (2005), submitted.

[13] Holzmann, G. J., "The Spin Model Checker," Addison-Wesley, 2003.

[14] Iosif, R., *Symmetry reductions for model checking of concurrent dynamic software*, Software Tools for Technology Transfer (STTT) **6** (2004), pp. 302–319.

[15] Khurshid, S., C. S. Pasareanu and W. Visser, *Generalized symbolic execution for model checking and testing*, in: *Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2003, pp. 553–568.

[16] Marriot, K. and P. J. Stuckey, "Programming with Constraints," The MIT Press, 1998.

[17] Matthew B. Dwyer, M. H. R., John Hatcliff, *Building your own software model checker using the bogor extensible model checking framework*, in: *Proceedings of the 17th Conference on Computer-Aided Verification (CAV 2005)*, 2005, to appear.

[18] Musuvathi, M. and D. Dill, *An incremental heap canonicalization algorithm*, Technical Report MSR-TR-2005-37, Microsoft Research (2004).

[19] Robby, J. H., Matthew B. Dwyer, *Bogor: An extensible and highly-modular model checking framework*, in: *Proceedings of the Fourth Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003)*, 2003.

[20] Robby, J. H. R. I., M. B. Dwyer, *Space-reduction strategies for model checking dynamic software*, in: *Proc. SoftMC03 Workshop on Software Model Checking*, 2003.

[21] Testing, Verification and Measurement, Microsoft Research, *Zap theorem prover*, http://research.microsoft.com/tvm/.

[22] Tillmann, N., W. Schulte and W. Grieskamp, *Parameterized unit tests*, Technical Report MSR-TR-2005-64, Microsoft Research (2005), to appear in FSE 2005.

THROW l	RETHROW	UNWIND
BRANCH $b, l^?$	CALLINDIRECT $l_1^? \leftarrow l_2(\bar{l})$	RETURN $l^?$
UNARY $l_1 \leftarrow \oplus l_2$	BINARY $l_1 \leftarrow l_2 \otimes l_3$	CALLPRIMITIVE $p(\bar{l})$
NEWOBJECT $l \leftarrow \tau$	NEWARRAY $l_1, \tau[l_2]$	ISINSTANCE $l_1 \leftarrow l_2, \tau$
LOADEXCEPTION l	LOADCONST $l \leftarrow c$	LOADLOCALADDR $l_1 \leftarrow l_2$
LOADFIELDADDR $l_1 \leftarrow l_2^?.f$	LOADELEMENTADDR $l_1 \leftarrow l_2[l_3], \tau^?$	LOADELEMENT $l_1 \leftarrow l_2[l_3]$
STOREELEMENT $l_1[l_2] \leftarrow l_3$	LOADMETHODADDR $l_1 \leftarrow l_2^?.m$	LOADINDIRECT $l_1 \leftarrow l_2$
STOREINDIRECT $l_1 \leftarrow l_2$		

Fig. A.1. Non-redundant subset of XIL

A XIL's Instruction Set

Figure A.1 introduces XIL's non-redundant instruction set. We write l for a local, $\bar{l}$ for a sequence of locals, and $l^?$ for a local which is optional in the context of an instruction. For example, the branch instruction has an optional local; depending on the value of the local execution continues at the label. If no local given, the jump is unconditional. We use c to denote a constant value, m to denote a method, f to denote a field, and τ to denote a .NET type. $\oplus$ and $\otimes$ range over unary and binary operations.

Figure A.1 omits XIL instructions which can be also expressed using their address-based counter parts: for example, loading the value of a field can be expressed by loading the address of the field and then performing an indirect load on that address. Only for loading and storing array element, separate instructions are necessary for checking array co-variance. Note that addresses are fundamental in order to express .NET concepts like delegates and reference parameters.

Loading of static fields and instance field addresses is expressed by the same load-field-address instruction, where the instance is optional. Note that CIL unifies the handling of struct (.NET value type) fields with those of object fields by using struct addresses where object references can occur, which we adapted also for XIL.

Most instructions should go without saying, but some require explanation. NEWOBJECT $l \leftarrow \tau$ creates a raw object; a call to the constructor, which is treated like an instance method, must follow immediately. LOADMETHODADDR $l_1 \leftarrow l_2^?.m$ uses the runtime type of the optional $l_2^?$ to perform a virtual method lookup (or raises an exception if $l_2^?$ is present but **null**); if $l_2^?$ is omitted, then the address of the given method will be taken. Note that a method address is really just the address, and does not aggregate the receiver object (there is no one-to-one mapping between .NET delegates and method addresses, but the latter are used to implement the former). Therefore, on CALLINDIRECT $l_1^? \leftarrow l_2(\bar{l})$ the receiver must be supplied as the first element of $\bar{l}$ if the method address l_2 points to an instance method. Taking the address of an array element or storing an array element performs the following checks because of array co-variance, in addition to the usual null-dereference and index-out-of-bounds checks. LOADELEMENTADDR $l_1 \leftarrow l_2[l_3], \tau^?$ requires a

type argument τ if the array's element type is a reference type. In this case, this instruction checks if the array's element type is equal to τ. If it is not, an `ArrayTypeMismatchException` is raised. The `STOREELEMENT` $l_1[l_2] \leftarrow l_3$ instruction checks if l_3 is assignable to the array's element type. If it is not, an `ArrayTypeMismatchException` is raised.

When an exception is raised, either explicitly by the `THROW` instruction or as a side effect of another instruction, the exception object is stored as the current exception in the state, and control flow continues with the exception exit block of the current block in the control flow graph. The instruction `LOADEXCEPTION` l retrieves the current exception object (or **null** if there is no current exception) and stores it in l. This instruction is usually followed by a type check on l, followed by a `RETHROW` instruction when the type check fails. The current exception is reset to **null** when a method is left normally with the `RETURN` instruction. If a method is left with the `UNWIND` instruction, the current exception will be raised in the caller.

The instruction `CALLPRIMITIVE` $p(\bar{l})$ is the extension point of XIL. Here p denotes a delegate (in the XRT framework, i.e. on the meta level) which is called by the instruction stepper for interpreting this instruction.

Compared to source languages like C#, simplifications in the type system of XIL apply; these simplifications arise from the way CIL instructions interact with the memory and the evaluation stack. Types like booleans, characters, and signed and unsigned integers up to 32 bits are collapsed into one type, int32. Accordingly the other primitive types are collapsed into int64 and double. Arithmetic instructions like addition always operate on the collapsed types. The semantics of the original types is preserved by according widening and narrowing conversions. In CIL, these conversions are mostly implicit in load and store instructions; in XIL the conversions are always explicit instructions.

We say two types are *representation compatible* if they either map to the same collapsed primitive type (int32, int64, or double), or if they are both of the same "struct" type, or if they are both reference types, or if they are both address types which refer to representation compatible locations, or if they are both method addresses with representation compatible signatures. XIL– which originates from safe, verified CIL– ensures the following static typing assumptions:

- if a field address is selected from an object or struct, this field actually exists;
- if a virtual method address is loaded, the method actually exists;
- if an element address is selected, the object is actually an array;
- only representation compatible values are moved from one location to another;
- if a method is called indirectly, the addressed method's signature is representation compatible with the arguments and result provided by the call.

B Sample: Exploring Random Choice

We give a code sample which shows how to write a simple explorer of random selections in an arbitrary .NET program. Figure B.1 shows how to perform a type

```
partial class RandomChoiceExplorer : ApplicationBase {
  IMethodImplementation Setup(){
    IProgram program = this.GetRequiredService<IProgram>();
    IAssembly assembly =
      program.LoadAssemblyFrom(this.Configuration.ProgramName);
    IMethodFlavor methodFlavor = program.CreateExecutionFlavor();
    ISubstitutedType randType = program.SystemAssemblies.MSCorLib
                                .SubstituteType("System.Random");
    randType.DefineMethod(
        "System.Random.Next(System.Int32,System.Int32)",
        methodFlavor,new PrimitiveCall.Handler(this.Handle));
    return assembly.EntryPoint.GetImplementation(methodFlavor);
  }
  ISuspension Handle(IActiveState state, ILocal[] locals){
    ILocal result = locals[0]; ILocal inst = locals[1];
    ILocal min = locals[2]; ILocal max = locals[3];
    if (!state.GetAndClearCurrentThreadReExecution()){
      // called first time
      int minValue = state.GetInt32(state.GetLocal(min));
      int maxValue = state.GetInt32(state.GetLocal(max));
      state.SetCurrentThreadReExecution();
      return new RandomChoice(
                  state.TransactionContext,state.Compress(),
                  state.CurrentThread,minValue,maxValue);
    } else {
      // called in re-execution mode
      RandomChoice rc =
           (RandomChoice)state.TransactionContext.Suspension;
      state.SetLocal(result,state.MakeInt32(
            rc.min + state.TransactionContext.Index));
      return null;
    }
  }
  class RandomChoice : SuspensionBase {
    int min;
    RandomChoice(Transaction context, ICompressedState cstate,
                 int currentThread, int min, int max)
      : base(context,cstate,currentThread,max-min+1) {
      this.min = min;
    }
    override public ISuspension ComputeTransaction(int index){
      IActiveState astate =
         this.State.Uncompress(this.CurrentThread);
      return this.Executor.RunUntilNextSuspension(
                 astate,new Transaction(this,index));
    } } }
```

Fig. B.1. Random Choice Explorer: type substitution and custom suspension

substitution, how to write a primitive call handler which uses re-execution, and how to define a custom suspension. Figure B.2 shows code which performs a simplistic state space exploration. The sample is complete and doesn't omit any details needed to make it run.

The class `RandomChoiceExplorer` in Figure B.1 derives from an XRT class `ApplicationBase` which facilitates the construction of applications. This class provides command line parsing and a standard setup of the components of XRT, and since it is a component itself, allows querying services using `GetRequiredService`, and the `Configuration` object available to each component.

The method `Setup` in Figure B.1 performs the type substitution and returns the top-level entry point of the program being explored. After querying the `IProgram`

```
partial class RandomChoiceExplorer  {
  static void Main(string[] args){
    RandomChoiceExplorer x = new RandomChoiceExplorer();
    x.InstallFromConfiguration("xrtrand", args);
    x.Explore(x.Setup());
  }
  void Explore(IMethodImplementation entryPoint){
    Dictionary<ICompressedState,bool> visited =
      new Dictionary<ICompressedState,bool>();
    Queue<Transaction> frontier = new Queue<Transaction>();
    ICompressedState initialState =
      this.GetRequiredService<IStateProvider>.InitialState;
    IExecutor executor =
      this.GetRequiredService<IExecutorFactory>
            .CreateExecutor();
    ISuspension susp =
       executor.Activate(initialState,Values.MainThread,
                         entryPoint,null);
    do {
      visited[susp.State] = true;
      bool terminal = true;
      foreach (Transaction t in susp.Transactions){
        frontier.Enqueue(t); terminal = false;
      }
      if (terminal && !(susp is ITerminationSuspension))
         Console.WriteLine("deadlock!");
      susp = null;
      while (frontier.Count > 0){
        Transaction next = frontier.Dequeue();
        ISuspension s =
          next.Suspension.ComputeTransaction(next.Index);
        if (!visited.ContainsKey(s.State)){
           susp = s; break;
        }
      }
    } while (susp != null);
  } }
```

Fig. B.2. Random Choice Explorer: main entry point and exploration algorithm

component which provides the program model, `Setup` loads the program assembly. It then performs a type substitution for `System.Random`. The only method we are going to define in the substituted `System.Random` is `int Next(int min, int max)`. The original method delivers a random value in the range `min` to `max`.

The method `Handle` is the primitive call handler which substitutes `System.Random.Next`. It determines whether it is executed the first time, and in this case creates a suspension which represents the choice point for random selection. Otherwise it calculates the value to deliver for the random choice in the current transaction and stores it in the *result* parameter. The property `state.TransactionContext` returns the transaction in which context this call handler is executed. A transaction is given by the suspension which has created it, and an index identifying a particular transaction outgoing from that suspension. In the simple case of the random choice, we can just use the index to calculate the result value.

The class `RandomChoice` realizes the custom suspension. It derives from `SuspensionBase`, which implements the interface `ISuspension`. The

$$
\begin{array}{lll}
t ::= x \mid o \mid i \mid \tau & & \textit{variables, objects, integers, types} \\
\quad \mid \quad \oplus t \mid t_1 \otimes t_2 & & \textit{unary and binary} \\
\quad \mid \quad \tau(t) \mid t_1 \preceq t_2 \mid m_t & & \textit{typing, subtyping, method selection} \\
\quad \mid \quad [:= t]_f \mid t_1[t_2 := t_3]_f \mid t_1[t_2]_f & & \textit{field maps} \\
\quad \mid \quad \{t\} \mid t_1 \mathinner{..} t_2 \mid t_1 \cup t_2 \mid \theta(|t|) \mid \{:= t\}_f \mid t_1 \in t_2 & & \textit{domains}
\end{array}
$$

Fig. C.1. Representative subset of the term language

base class constructor is passed the current transaction context, the compressed state, the current thread, and the number of transactions. For illustration purposes, we have overridden one of the methods of `SuspensionBase`, namely `ComputeTransaction`, though the overridden version does exactly the same as in the base class: it uncompresses the state of the suspension and then calls an auxiliary method of the executor which runs the instruction stepper until the next suspension is hit.

In Figure B.2 we define the entry point of the sample program, as well as the exploration algorithm. This algorithm is independent of the actual problem at hand, i.e. works with any kind of suspensions configured. It uses the method `executor.Activate` to create an initial activation suspension for the entry point method of the loaded program. It then performs a breadth-first exploration. During exploration, it performs a simple deadlock check by only allowing a special suspension which represents thread termination to be terminal. The termination suspension is created by the executor upon return from a top-level method of the current thread.

C Terms

The grammar of the term language is given in Figure C.1. This grammar omits terms for numeric values other than integers, "struct" values, and arrays. These omitted terms do not bring conceptual surprises. Also, we simplified field map domains. In the following we discuss some aspects of the term representation which might not be obvious. τ represents a type *value*, i.e. a ground term representing a type. $\tau(t)$ denotes the runtime type of the object given by t. $t_1 \preceq t_2$ represents a subtype assertion, where t_1 and t_2 are terms denoting types. The method designator term m_t, where m denotes a method name, represents the *address* of m in the type t; when m is a virtual method, m_t represents a virtual method lookup. This is used to represent a symbolic result of the `LOADMETHODADDR` $l_1 \leftarrow l_2.m$ instruction. We say that m_τ is the *normal* method designator term of m if τ is the declaring type of m.

We take a closer look at field map terms. A field map logically represents a mapping from objects to field assignments. This mapping is syntactically given by a series of updates on an initial field map. If we have a *closed world*, the initial field map is denoted by the term $[:= t]_f$. In this field map, every object has the assigned

value t, which usually represents the default value of the according type of the field. In an *open world*, the initial field map is a free logical variable.

Updates on field maps are described by terms $t_1[t_2 := t_3]_f$, where t_1 is the field map, t_2 the object, and t_3 the assigned value. Selections on field maps are given as $t_1[t_2]_f$, where t_1 is the field map and t_2 is the object. We will omit the f index where it is clear from the context. We can reduce selections under certain conditions. Let m be a field map. Consider the term $m[o_1 := t_1][o_2 := t_2][o_1]$, where $o_1 \neq o_2$. This reduces to t_1. Compare with $m[o_1 := t_1][x := t_2][o_1]$, where x is an unbound variable. This selection term cannot be reduced since the update on x can address any object, including o_1.

Domain terms are represented by singletons, $\{t\}$, ranges, $t_1 \,..\, t_2$, union, $t_1 \cup t_2$, projection $\theta(\!|t|\!)$, membership, $t_1 \in t_2$, and field map domains, written $\{:= t\}$.

A field map domain represents a set of field maps by a domain of their assigned values. Consider the singleton domain $\{t\}$; then the field map domain $\{:= \{t\}\}$ could be the domain of any field map in which every contained object has the assigned value t.

Projections are a specialized form of set comprehension which is used in derived domain calculations. Here, θ is a function mapping terms into terms. The reduction rules for projections are

$$\begin{array}{ll} \theta(\!|\{t\}|\!) & = \{\theta(t)\} \\ \theta(\!|t_1 \cup t_2|\!) & = \theta(\!|t_1|\!) \cup \theta(\!|t_2|\!) \\ (\_[\_])(\!|\{:= t\}|\!) & = t. \end{array}$$

where the operator $(\_[\_])$ denotes arbitrary object field selection.

We sketch the computation of derived domains. They are essential for our ability to split assumption sets, e.g. to resolve virtual method calls with symbolic target receivers. Recall that domains represent *approximations* of the actual value of a term, which might be further restricted by other constraints in the assumption store.

In Figure C.2, $\mathsf{dom}(t)$ delivers the domain of a term, which is either explicitly given in the assumption set, or derived by $\mathsf{dom}_d(t)$. The derived domain computation proceeds as follows:

- If a variable term does not have an explicit assumption about its domain, we create a new variable which represents the domain and add this to the assumption set.
- If a term denotes a constant value, then the derived domain is the singleton of the value.
- For binary and unary terms, the derived domain is constructed taking the domain of the sub-terms and the semantics of the according operator into account.
- Determining the derived domain of the subtype relation is overapproximated by simply saying that it can be **false** or **true**, here denoted by 0 or 1.
- For runtime type denoting terms, the derived domain is computed by mapping the runtime type to the result of the domain computation of the embedded term, i.e. assume the assumption store contains $x \in \{o_1\} \cup \{o_2\}$, then $\mathsf{dom}_d(\tau(x)) =$

$$
\begin{array}{ll}
\mathsf{dom}(t) & = \textbf{if the assumption set contains}(t \in D)\ \textbf{then}\ D\ \textbf{else}\ \mathsf{dom}_d(t) \\
\mathsf{dom}_d(x) & = x' \text{ where } x' \text{ is fresh and } x \in X' \text{ is added to the assumption set} \\
\mathsf{dom}_d(o) & = \{o\} \\
\mathsf{dom}_d(i) & = \{i\} \\
\mathsf{dom}_d(\tau) & = \{\tau\} \\
\mathsf{dom}_d(\oplus\, t) & = \text{derive domain from } \mathsf{dom}(t) \text{ and semantics of } \oplus \\
\mathsf{dom}_d(t_1 \otimes t_2) & = \text{derive domain from } \mathsf{dom}(t_i) \text{ and semantics of } \otimes \\
\mathsf{dom}_d(t_1 \preceq t_2) & = 0\,..\,1 \\
\mathsf{dom}_d(\tau(t)) & = (\lambda\sigma.\tau(\sigma))(\!|\mathsf{dom}(t)|\!) \\
\mathsf{dom}_d(m_t) & = (\lambda\sigma.m_\sigma)(\!|\mathsf{dom}(t)|\!) \\
\mathsf{dom}_d([:= t]) & = \{:= \{t\}\} \\
\mathsf{dom}_d(t_1[t_2 := t_3]) & = \mathsf{dom}(t_1) \cup \{:= \{t_3\}\} \\
\mathsf{dom}_d(t_1[t_2]) & = (\_[\_])(\!|\mathsf{dom}(t_1)|\!)
\end{array}
$$

Fig. C.2. Derived domain computation

$\{\tau(o_1)\} \cup \{\tau(o_2)\}$.

- The derived domain of a method designator is similarly computed by mapping the method designator to the elements of the domain of the embedded term, i.e. assume the assumption store contains $x \in \{\tau_1\} \cup \{\tau_2\}$, then $\mathsf{dom}_d(m_x) = \{m_{\tau_1}\} \cup \{m_{\tau_2}\}$.
- Derived field map domains are constructed from the initial field map value and subsequently assigned values. Selecting an element from a field map is then realized using a projection term, for instance $\mathsf{dom}_d([:= t_1][t_2 := t_3][t_2]) = \{t_1\} \cup \{t_3\}$.

D Instruction Rewriting for Symbolic Exploration

In this section, we illustrate how symbolic exploration is realized in $\text{XRT}_\mathcal{S}$ using instruction rewriting on XIL as defined in Appendix A. $\text{XRT}_\mathcal{S}$ uses *primitive calls* to introduce new instructions. New instructions are either checkpoint or replacement instructions. We adorn new instructions with the suffix $\mathcal{S}$ to distinguish them from the original XIL instruction set.

D.1 Checkpoint Instructions

The purpose of checkpoint instructions is to concretize symbolic values just as much as necessary to allow often implicit control-flow decisions to be made. For example, loading an instance field with a **null** receiver will raise an exception. Thus, if a symbolic receiver value can be **null**, this case must be split from the non-**null** case, where – although the precise receiver object may still remain unknown – a symbolic term representing the access can always be constructed sucessfully.

Case splits are realized using the suspension mechanism of XRT as follows. When a checkpoint instruction is executed, it creates a particular term which represents the value on which the case split depends. It then uses assumption split over that term (which in turn uses term domains as described in Appendix C) to

$\texttt{GROUND}_{\mathcal{S}}\, l_1 \leftarrow l_2$	$\texttt{CHECKNULL}_{\mathcal{S}}\, l_1 \leftarrow l_2$	$\texttt{CHECKRANGE}_{\mathcal{S}}\, l_1 \leftarrow l_2, l_3$
$\texttt{CHECKNONNEG}_{\mathcal{S}}\, l_1 \leftarrow l_2$	$\texttt{CHECKUNARY}_{\mathcal{S}}\, l_1 \leftarrow \oplus l_2$	$\texttt{CHECKBINARY}_{\mathcal{S}}\, l_1, l_2 \leftarrow l_3 \otimes l_4$
$\texttt{CHECKELEMENTTYPE}_{\mathcal{S}}\, l_1, l_2^?, \tau^?$		

Fig. D.1. Checkpoint instructions

$\texttt{UNARY}_{\mathcal{S}}\, l_1 \leftarrow \oplus l_2$	$\texttt{BINARY}_{\mathcal{S}}\, l_1 \leftarrow l_2 \otimes l_3$
$\texttt{LOADMETHODADDR}_{\mathcal{S}}\, l_1 \leftarrow l_2.m$	$\texttt{ISINSTANCE}_{\mathcal{S}}\, l_1 \leftarrow l_2, \tau$

Fig. D.2. Replacement instructions performing symbolic computations

$\texttt{RETHROW}$	$\texttt{UNWIND}$	$\texttt{RETURN}\, l^?$
$\texttt{CALLPRIMITIVE}\, p(\bar{l})$	$\texttt{NEWOBJECT}\, l \leftarrow \tau$	$\texttt{LOADEXCEPTION}\, l$
$\texttt{LOADCONST}\, l \leftarrow c$	$\texttt{LOADLOCALADDR}\, l_1 \leftarrow l_2$	$\texttt{LOADINDIRECT}\, l_1 \leftarrow l_2$
$\texttt{STOREINDIRECT}\, l_1 \leftarrow l_2$	$\texttt{BRANCH}\, b, -$	$\texttt{LOADFIELDADDR}\, l_1 \leftarrow -.f$
$\texttt{LOADMETHODADDR}\, l \leftarrow -.m$		

Fig. D.3. Instructions unaffected by $\text{XRT}_{\mathcal{S}}$ instruction rewriter. Note that some instructions are versions with absent optional locals.

original instruction	new instructions
$\texttt{THROW}\, l$	$\texttt{CHECKNULL}_{\mathcal{S}}\, l^* \leftarrow l;\ \texttt{THROW}\, l^*$
$\texttt{BRANCH}\, b, l$	$\texttt{GROUND}_{\mathcal{S}}\, l^* \leftarrow l;\ \texttt{BRANCH}\, b, l^*$
$\texttt{CALLINDIRECT}\, l_1^? \leftarrow l_2(\bar{l})$	$\texttt{GROUND}_{\mathcal{S}}\, l^* \leftarrow l_2;\ \texttt{CALLINDIRECT}\, l_1^? \leftarrow l^*(\bar{l})$
$\texttt{NEWARRAY}\, l_1, \tau[l_2]$	$\texttt{CHECKNONNEG}_{\mathcal{S}}\, l^* \leftarrow l_2;\ \texttt{NEWARRAY}\, l_1, \tau[l^*]$
$\texttt{LOADFIELDADDR}\, l_1 \leftarrow l_2.f$	$\texttt{CHECKNULL}_{\mathcal{S}}\, l^* \leftarrow l_2;\ \texttt{LOADFIELDADDR}\, l_1 \leftarrow l^*.f$
$\texttt{LOADELEMENTADDR}\, l_1 \leftarrow l_2[l_3], \tau^?$	$\texttt{CHECKNULL}_{\mathcal{S}}\, l_1^* \leftarrow l_2;\ \texttt{CHECKRANGE}_{\mathcal{S}}\, l_2^* \leftarrow l_3, l_1^*$ $\texttt{CHECKELEMENTTYPE}_{\mathcal{S}}\, l_2^*, -, \tau^?$ $\texttt{LOADELEMENTADDR}\, l_1 \leftarrow l_1^*[l_2^*],$
$\texttt{LOADELEMENT}\, l_1 \leftarrow l_2[l_3]$	$\texttt{CHECKNULL}_{\mathcal{S}}\, l_1^* \leftarrow l_2;\ \texttt{CHECKRANGE}_{\mathcal{S}}\, l_2^* \leftarrow l_3, l_1^*$ $\texttt{LOADELEMENT}\, l_1 \leftarrow l_1^*[l_2^*]$
$\texttt{STOREELEMENT}\, l_1[l_2] \leftarrow l_3$	$\texttt{CHECKNULL}_{\mathcal{S}}\, l_1^* \leftarrow l_1;\ \texttt{CHECKRANGE}_{\mathcal{S}}\, l_2^* \leftarrow l_2, l_1^*$ $\texttt{CHECKELEMENTTYPE}_{\mathcal{S}}\, l_1^*, l_3, -$ $\texttt{STOREELEMENT}\, l_1^*[l_2^*] \leftarrow l_3$

Fig. D.4. Instruction rewriting introducing checkpoints.

original instruction	new instructions
$\texttt{UNARY}\, l_1 \leftarrow \oplus l_2$	$\texttt{CHECKUNARY}_{\mathcal{S}}\, l^* \leftarrow \oplus l_2;\ \texttt{UNARY}_{\mathcal{S}}\, l_1 \leftarrow \oplus l^*$
$\texttt{BINARY}\, l_1 \leftarrow l_2 \otimes l_3$	$\texttt{CHECKBINARY}_{\mathcal{S}}\, l_1^*, l_2^* \leftarrow l_2 \otimes l_3;\ \texttt{BINARY}_{\mathcal{S}}\, l_1 \leftarrow l_1^* \otimes l_2^*$
$\texttt{ISINSTANCE}\, l_1 \leftarrow l_2, \tau$	$\texttt{ISINSTANCE}_{\mathcal{S}}\, l_1 \leftarrow l_2, \tau$
$\texttt{LOADMETHODADDR}\, l_1 \leftarrow l_2.m$	$\texttt{CHECKNULL}_{\mathcal{S}}\, l^* \leftarrow l_2;\ \texttt{LOADMETHODADDR}_{\mathcal{S}}\, l_1 \leftarrow l^*.m$

Fig. D.5. Instruction rewriting introducing checkpoints and replacing instructions.

enumerate all the different assumption sets under which control flow could continue, filtering out those which are infeasible. We consider an assumption set as feasible for the split if it is either satisfiable or if satisfiablity is inconclusive. The checkpoint instruction creates a symbolic suspension whose outgoing transactions

represent the different assumptions under which control flow could continue. This process is lazy such that the assumptions and feasibilty checks are generated the time the next transaction is queried from the suspension.

Figure D.1 shows the checkpoint instructions. The semantics is as follows:

- $\texttt{GROUND}_{\mathcal{S}}\, l_1 \leftarrow l_2$ enumerates over all possible ground values of l_2 and places them in l_1.
- $\texttt{CHECKNULL}_{\mathcal{S}}\, l_1 \leftarrow l_2$ enumerates over all solutions of $l_2 =$ **null**. If the equation holds, it places **null** in l_1; otherwise it copies the value of l_2 into l_1.
- $\texttt{CHECKRANGE}_{\mathcal{S}}\, l_1 \leftarrow l_2, l_3$ enumerates over the solutions of the equation $0 \leq l_2 <$ len, where len is a term representing the length of the array object in l_3. If the constraint holds, l_2 is copied into l_1; otherwise -1 is placed in l_1.
- $\texttt{CHECKNONNEG}_{\mathcal{S}}\, l_1 \leftarrow l_2$ enumerates over the solutions of the equation $0 \leq l_2$. In case the constraint holds, l_2 is copied into l_1; otherwise -1 is placed in l_1.
- The instructions $\texttt{CHECKUNARY}_{\mathcal{S}}\, l_1 \leftarrow \oplus l_2$ and $\texttt{CHECKBINARY}_{\mathcal{S}}\, l_1, l_2 \leftarrow l_3 \otimes l_4$ check conditions specific for the particular operations. Consider for example the division l_3/l_4; all solutions of the equation $0 = l_4$ will be enumerated. If the equation holds, 0 is stored in l_2; otherwise, l_3, l_4 are copied into l_1, l_2.
- $\texttt{CHECKELEMENTTYPE}_{\mathcal{S}}\, l_1, l_2^?, \tau^?$ performs checks necessary because of array covariance. l_1 indicates the array, $l_2^?$ can indicate a value to be stored in the array, and $\tau^?$ can indicate the exact element type of the array. There are three cases. Case 1: l_1 is **null**; nothing happens. Case 2: l_1 is not **null** and l_2 is present. If l_2 is not not assignable to the element type of the array l_1, an `ArrayTypeMismatchException` is raised. Case 3: l_1 is not **null** and τ is present. If the element type of the array l_1 is not equal to τ, `ArrayTypeMismatchException` is raised.

D.2 Replacement Instructions

Figure D.2 shows new instructions which will serve as replacements for instructions which perform concrete computations. If all arguments are concrete, the replacement instructions will perform concrete computations as well; however, if some arguments are symbolic, these instructions construct new symbolic terms representing the result of the computation.

$\text{XRT}_{\mathcal{S}}$ employs an instruction rewriter to modify or replace instructions. Each instruction is transformed according to one of the following three cases:

- The instructions listed in Figure D.3 remain unchanged.
- Figure D.4 illustrates how checkpoint instructions are inserted as guards before certain instructions, such that the instruction operates on arguments preprocessed by the checkpoint. In the process, auxiliary locals of the form l^* are introduced.
- The remaining instructions are replaced entirely. See Figure D.5 for details.